Universidad
Carlos III de Madrid

Universidad Carlos III de Madrid

Trabajo de Fin de Grado

Diseño de una solución de movilidad de cliente basada en SDN

Student:
Pablo Castellanos
Santamaría

Tutor:
Antonio De la Oliva

June 22, 2014

Agradecimientos

Después de estos años de esfuerzo y sacrificio tengo que agradecer a muchas personas el haber logrado superar el reto de terminar el Grado en Ingeniría Informática. Esto no habría sido posible sin la ayuda de mis profesores y compañeros de carrera, que durante estos años han hecho más llevadero el camino. Por otro lado, he tenido la gran suerte de haber contado con el apoyo de Antonio, mi tutor y la persona que ha hecho posible este proyecto; y de Isabel, la persona que me ha despejado el camino cuando han surgido las dificultades. Sin su ayuda dificilmente podría haberlo conseguido. Por último quiero agradecer a las personas más importantes de mi vida, las que apostaron porque este día llegaría y gracias a las cuales he sacado las fuerzas necesarias para terminar esta aventura: Esther, Paloma y Mayka.

Si un hombre atraviesa un río
y una barca vacía
choca con su propia barca,
no se ofenderá ni se pondrá furioso
por muy mal genio que tenga.
Pero si la barca va tripulada,
se enardece acaso, gritando y maldiciendo,
sólo porque ve a un remero.
Advierte que todas las barcas están vacías
mientras atraviesas el río del mundo,
y nada será capaz de ofenderte.
Proverbio Zen

Contents

List of Figures

List of Tables

Abstract

Due to the complex scene of the growing networks infrastructure in the big cities, and because of the increasing amount of the mobile devices, the management of the users' flows has become into a big challenge for the network operators. Taking these issues into consideration, the European Union has developed one line of research within the framework of the CROWD project to develop a solution for high density networks and the issue of the users mobility inside this kind of networks.

The main goal of this final degree project is to create a control program which, through the software defined network (SDN) paradigm, implements the mobility management of a mobile node, for the cases of network- and host-based mobility. The mobility approach of this project has two different points of view. On the one hand we have implemented a mobility management solution which relies on the network to take care of the mobility of the user. In this network-based scene, the actions to ensure the complete connectivity during the transit of the mobile node amongst two districts are leaded by two different actors, the CROWD Regional Controller (CRC) and the CROWD Local Controller (CLC). This approach is completely transparent to the user attributing all the operations to the controllers and allowing the mobility of the user within the network without doing any specific action.

On the other hand, we have the host-based mobility approach. This design also includes the participation of the mobile node during the handover interactions. Its participation consists on creating an IPv6 in IPv4 tunnel between the public IPv4 of the Mobile Node and the public IPv4 of the Default Gateway with which it is anchored to. Mobile Node has to be able to communicate with the CROWD Regional Controller.

With these two different approaches, the project aims to create a software solution for the scenario of extremely dense networks

CHAPTER 1

Introduction

The main goal of this final degree project is to develop an application or controller based on the SDN [1] paradigm capable of implementing distributed mobility management (DMM) [2] within the context of the CROWD EU FP7 project [1]. Taking the problem of ultra dense future networks into consideration, Distributed Mobility Management (DMM) working group (WG) of the IETF [2] has presented different solutions to solve the mobility issue. The main idea is to swap from current hierarchical network architectures to another paradigm based on decentralized entities.

To achieve these mobility objectives we will use the approach of SDN due to its benefits: low economic costs, high tunning capabilities and separation of control and data planes. A further description of the possibilities brought by the use of SDN can be found on the State of the Art Chapter 2.2.

The communications protocol selected that gives access to the forwarding plane of a network switch over the network is OpenFlow[3]. OpenFlow allows remote administration of a switch's packet forwarding tables, by adding, modifying and removing packet matching rules and actions. Thanks to that, routing decisions can be made ad hoc by the controller of the network. Packets which are unmatched by the switch are forwarded to the controller and depending of the type, source and destination of each packet, the controller decides to modify, install or delete existing flow table rules on one or more OpenFlow controlled switches.

There are several Open Flow controllers available to manage the approach set out by the DMM study group. For this project we have chosen Ryu as an Open Flow controller due to its advantages discussed on the Solution Proposed, Chapter 3. The software will manage all the connections and traffic through an IPv6 [4] network. We will design an implementation of the scenario envisioned in the CROWD project with two districts managed by a CROWD Regional Controller and internally managed by a CROWD Local Controller. Once we have decided the network design we will implement a testbed to test the performance of the mobility management solution. This testbed will include some access points, open flow nodes, openflow controllers and a

[1] `http://www.ict-crowd.eu/`

[2] `http://datatracker.ietf.org/wg/dmm/charter/`

mobile node capable to perform different mobility handovers focused on assessing the viability solution proposed by the DMM study group. Furthermore, the mobile node will be able to perform a handover from our IPv6 network to a different IPv4 [5] network.

Finally on the Experiments (Chapter 5) we will do different tests to completely evaluate the performance of the SDN deployed and to check if the present solution is capable of managing the mobility of users and their performance.

1.1 Analysis of the problem

Current networks architectures are deployed in a hierarchical manner, relying on a centralized gateway. Thus, the existing IP mobility management protocols are generally deployed in a centralized manner. All the data traffic passes through a centralized mobility anchor, such as the HA in MIPv6 [6] or the LMA in PMIPv6 [7], and all the bindings are managed at this anchor as well. As the number of MNs and the volume of the mobile data traffic increase, such centralized architectures may encounter scalability issues (e.g. network bottlenecks, and single point of failure), security issues (e.g. attacks focused on the centralized anchor), as well as performance issues (e.g. centralized and non-optimal routing). In addition, existing IP mobility protocols are designed to be always activated, managing all the services and all the traffic in the same way. They do not take into consideration that a given MN may not move during the use of a service (which is 60% of the cases in operational networks) or that a service may not require mobility functions at all. Such approaches may thus lead to non-optimal routing and large overhead due to tunnelling mechanisms.

In order to cope with the rapid traffic explosion we are witnessing, IP mobility management protocols need to be adapted for such evolution. There is a need to define novel mobility management mechanisms that are both distributed and offered dynamically. They should be distributed in order to avoid any network bottleneck or single point of failure, and to provide better reliability. They should be activated/deactivated dynamically as needed, in order to globally reduce their signalling load and to increase the achieved performance. Accordingly, the IETF chartered recently the DMM working group in 2012. Various efforts from both industry and academia are being performed on specifying DMM schemes. A common feature between different DMM schemes is distributing the mobility anchoring at the AR level. The MN changes dynamically its mobility anchor for new sessions, while keeping the previous anchors of ongoing sessions. When the sessions anchored at a specific mobility anchor are terminated, the MN unregisters from that anchor. Assuming that most of the sessions are relatively short, most of the data traffic is routed optimally without tunnelling. One of the DMM requirements is to rely on the existing IP mobility protocols by extending and adapting them. This is in order to benefit such standardized protocols before specifying new ones, and also to facilitate the migration of networks architectures.

Thus hereafter, we will consider two main approaches. The first is MIPv6-based [6], providing global as well as local mobility support for MNs that may move between several access networks. This protocol is designed to allow mobile device users to move from one network to another while maintaining a permanent IP address, allowing

nodes to remain reachable while moving around in the IPv6 Internet. To support these operations, Mobile IPv6 defines a new IPv6 protocol and a new destination option. All IPv6 nodes, whether mobile or stationary, can communicate with mobile nodes. The second protocol is PMIPv6-based [7], providing local mobility support for MNs moving in a single operational domain. It is a protocol for building a common and access technology independent of mobile core networks. Proxy Mobile IPv6 is the only network-based mobility management protocol standardized by IETF. A further explanation of protocols MIPv6 2.1.1 and PMIPv6 2.1.2 can be found on the State of the Art (Chapter 2)

CHAPTER 2

State of the art

2.1 Distributed Mobility Management

Distributed Mobility Management (DMM) is a new architectural paradigm for evolving mobile IP networks. The increase of mobile data demand from users due to a great variety of applications introduces additional challenges to mobility management. The huge data demand generated by mobile users has triggered the development of new paradigms which aim to flatter the network and distributing the entities in charge of managing users' mobility.

Current IP mobility protocols rely on a centralized and hierarchical architecture, which poses several critical issues as the following:

- All the traffic of the network goes throw a **Mobility Anchor**. This means scalability issues as bottlenecks, security issues, a centralized point of failure and issues related to distribute traffic between data and control.
- Current networks are always activated which leads to a non-optimal solution for managing users' mobility because mobile nodes are not always performing handovers

Mobility management schemes standardized by IETF for IPv6 networks are extensions to or modifications of the well known Mobile IPv6 protocol (MIPv6) [6], and can be classified into two main families: host-based mobility protocols and network-based mobility protocols.

- **Host-based** mobility approaches, such as MIPv6 and Dual Stack Mobile IPv6 (DSMIPv6) [8], enable global reachability and session continuity by introducing the Home Agent (HA), an entity located at the home network of the Mobile Node (MN) which anchors the permanent IP address used by the The MN, called the Home Address (HoA). The HA is in charge of defending the MNs HoA when the MN is not at home, and redirecting received traffic to the MNs' current location. When away from its home network, the MN acquires a temporal IP address from the visited network – called Care-of Address (CoA) – and informs the HA about its current location. An IP bi-directional tunnel between the MN and the HA is then used to redirect traffic to and from the MN [2]. Figure 2.1 presents the main concepts of DMM host-based architectures:

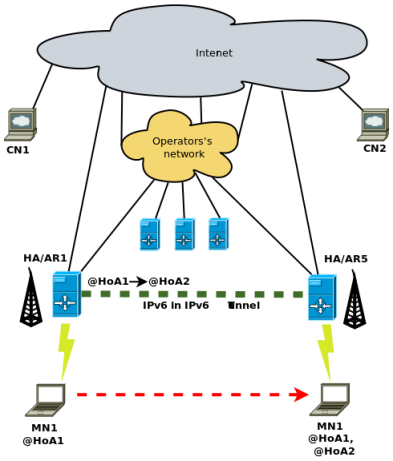


Figure 2.1: Host-Based DMM

- **Network-based** mobility approaches, such as Proxy Mobile IPv6 (PMIPv6) [7], provide MNs with mobility support without their involvement in mobility management and IP signaling, as the required functionality is relocated from the MN to the network. In particular, movement detection and signaling operations are performed by a new functional entity called the Mobile Access Gateway (MAG), which usually resides on the Access Router (AR) for the MN. In a Localized Mobility Domain (LMD), which is the area where the network provides mobility support, there are multiple MAGs. The MAG learns through standard terminal operation, such as router and neighbor discovery, or by means of link-layer support about an MNs movement and coordinates routing state updates without any mobility-specific support from the terminal [2]. Figure 2.2 presents the main concepts of DMM network-based architectures:

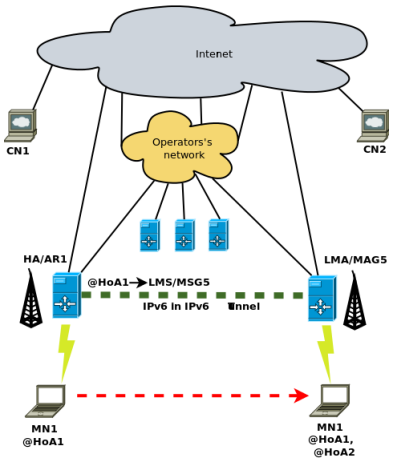


Figure 2.2: Network-Based DMM

Standards and research communities are working to address these issues within the new architectural paradigm, the Distributed Mobility Management (DMM). This paradigm place anchors closer to the user distributing the control and data infrastructures among the entities located at the edge of the access network. DMM also includes the ability for a mobile node to move between mobility anchors.

The DMM working group has been working in alternatives to *MIPv6* and *PMIPv6* to find a solution to these problems. Currently there is no consensus in the approach to be standardized as the DMM solution. There are four different approaches to to the distributed mobility management function:

1. **Peer-to-peer approaches:** Distribute mobility management functionality in various entities and to do it as much closer to the mobile node as possible.

2. **Architecture-dependent solutions:** Find solutions focused on providing enhanced mechanisms for local breakout and offloading to improve mobility, reducing the traffic through the operator core network. These proposals detect non-optimal paths.

3. **Solutions based on extending existing IETF protocols:** There are solutions designed based on the mobility behavior of *MIPv6* and *PMIPv6*. The ones adapting *MIPv6* encompass the use of Anchor Access Nodes (AAN) and Visited Access Nodes (VAN).

4. **Routing-based approaches:** Use routing updates between routers to manage the mobility within the mobility domain. The process is to detect the node's assigned prefix by the use of DNS lookups and then the exchange of Border Gateway Protocol (BGP) update messages to actualize the routing information. This proposal still lacks of deepness performance calculations.

Studying the mobility problem on dense networks within the scope of an SDN environment, we can isolate the forwarding issue into a unique element: the controller. It means that a very efficient solution can be proposed with minimum costs. Considering this, the main objective of this project is to implement a DMM-based solution through the application of the SDN paradigm.

Below we are going to explain the traditional mobility mechanisms: *MIPv6* and *PMIPv6*.

2.1.1 MIPv6

Mobile IPv6 [6][9] is a network protocol that allows nodes to remain reachable while moving around in the IPv6 Internet. MIPv6 enables global reachability and session continuity by introducing the HA, an entity located at the Home Network of the MN which anchors the permanent IP address used by the MN, called the HoA. The home agent is in charge of defending the mobile device's home address when it is not at home and redirecting received traffic to the mobiles' current location. When away from its home network, the mobile node acquires a temporal IP address from the visited network – called the CoA – and informs the HA about its current location by sending a Binding Update (BU) message. An IP bi-directional tunnel between the MN and the HA then redirects traffic to and from the mobile. In this way, the packets generated

by the mobile node's communication peer - called the CN - and sent to the mobiles' permanent address (i.e., its HoA) are tunnelled to the current MN location and arrive at the CoA. There is also optional support to avoid this suboptimal routing, which enables the MN to directly exchange traffic with a correspondent node without traversing the home network. This additional support is called Route Optimization and allows the mobile to inform correspondent nodes about its current location.

2.1.2 PMIPv6

PMIPv6 [7] is a mobility-management protocol that allows legacy mobile terminals to perform handover operations across heterogeneous networks, without their involvement in the management of their own IP mobility signalling. As an example of operation (see Fig. 2.3), consider a Localized Mobility Domain (LMD) scenario, where PMIPv6 provides mobility support, that comprises two MAGs, and an LMA. In addition to maintaining the state regarding the location of the MN in the LMD, the LMA must maintain an IPv6-in-IPv6 tunnel with every MAG for forwarding the data traffic of their MNs. When an MN first arrives at the LMD, it attaches to an Access Point (AP) and sends a RS message requesting an IPv6 prefix.

This message is received by the MAG, which asks the LMA for an IPv6 prefix for the MN through a PBU message. Next, the LMA replies to the MAG with a newly assigned IPv6 prefix for the MN through an Proxy Binding Acknowledgement (PBA) message and stores the mapping in its local lookup table, named Binding Cache (BC). Then, the MAG forwards the IPv6 prefix to the MN through an Router Advertisement (RA) message. Finally, the LMA uses the existing IPv6-in-IPv6 tunnel with the MAG (or creates a new one if there is none) for the data traffic exchanged by the MN with the network. When the MN moves to the coverage area of a second MAG, the process is repeated, but this time the LMA finds an existing entry in its BC for that MN, and therefore replies to the MAG with the same IPv6 prefix that the MN was using previously, updating the record for the MN and diverting its traffic to the new MAG tunnel. Thanks to the fact that the MAGs show the same layer-2 and IPv6 link local addresses to the MNs, these do not detect any layer-3 change while moving within the LMD. In conclusion, thanks to PMIPv6, a Mobile Node may change from one layer-2 Point of Attachment (PoA) to another, but it always keeps the same IP address across the LMD managed by an LMA. It is also worth noticing that the operation of PMIPv6 does not require the MN to implement any modification or extra software in its layer-3 stack, although it may require the assistance of some layer-2 mechanisms to work more efficiently.

2.2 Software Defined Network

Software Defined Network (SDN) is an approach to computer networking which evolved from work done at UC Berkeley and Stanford University around 2008. SDN allows network administrators to manage network services through abstraction of lower level functionality [1].This is done due to the physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices.

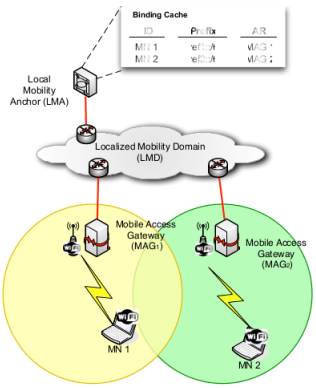


Figure 2.3: PMIPv6

SDN requires some method for the control plane to communicate with the data plane. One such mechanism, OpenFlow, is a foundational element for building SDN solutions and is often misunderstood to be equivalent to SDN, but other mechanisms could also fit into the concept.

Open Networking Foundation (ONF) [10] is a user-driven organization dedicated to the promotion and adoption of SDN, and implementing SDN through open standards where such standards are necessary to move the networking industry forward. The Open Networking Foundation was founded to promote SDN and OpenFlow, marketing the use of the term cloud computing before it became popular.

As the Open Networking Foundation says, Software-Defined Networking (SDN) is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. This architecture decouples the network control and forwarding functions, enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services.

The SDN architecture is:

- **Directly programmable:** Network control is directly programmable because it is decoupled from forwarding functions.
- **Agile:** Abstracting control from forwarding lets administrators dynamically adjust network-wide traffic flow to meet changing needs.
- **Centrally managed:** Network intelligence is (logically) centralized in software-based SDN controllers that maintain a global view of the network, which appears to applications and policy engines as a single, logical switch.
- **Programmatically configured:** SDN lets network manager to configure, manage, secure, and optimize network resources very quickly via dynamic, automated SDN programs, which they can write themselves because the programs do not depend on proprietary software.
- **Open standards-based and vendor-neutral:** SDN simplifies network design and operation because instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols.

2.2.1 OpenFlow

The necessity of the network researcher to run experimental protocols has lead the Open Networking Foundation to develop a new paradigm based on an Ethernet switch, with an internal flow-table, and a standardized interface to add and remove flow entries. This paradigm, named OpenFlow, allows researchers to run experiments on heterogeneous switches in a uniform way at line-rate and with high port-density. In addition, OpenFlow allows researchers to evaluate their ideas in real-world traffic settings.

The basic idea is to identify the common set of functions that run in many switches and to exploit these flow-tables. OpenFlow provides an open protocol to program the flow-table in different switches and routers. Researchers can control their own flows by choosing the routes their packets follow and the processing they receive. In this way, researchers can try new routing protocols, security models, addressing schemes, and even alternatives to IP.

To achieve the researcher objectives, its mandatory to install a concrete data-path for each kind of packet managed in the OpenFlow network. The data-path of an OpenFlow Switch consists of a Flow Table, and an action associated with each flow entry.

The OpenFlow Switch

The basic idea is simple: we exploit the fact that most modern Ethernet switches and routers contain flow-tables (typically built from TCAMs) that run at line-rate to implement firewalls, NAT, QoS, and to collect statistics. While each vendors' flow-table is different, we have identified an interesting common set of functions that run in many switches and routers. OpenFlow exploits this common set of functions. Figure 2.4 represents an OpenFlow Switch.

OpenFlow provides an open protocol to program the flow table in different switches and routers. A network administrator can partition traffic into production and research flows. Researchers can control their own flows - by choosing the routes their packets follow and the processing they receive. In this way, researchers can try new routing protocols, security models, addressing schemes, and even alternatives to IP. On the same network, the production traffic is isolated and processed in the same way as today. The

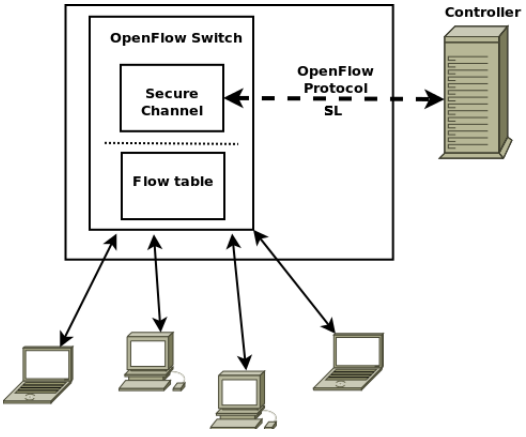


Figure 2.4: OpenFlow Switch

data-path of an OpenFlow Switch consists of a Flow Table, and an action associated with each flow entry. The set of actions supported by an OpenFlow Switch is extensible, but below we describe a minimum requirement for all switches. For high-performance and low-cost the data path must have a carefully prescribed degree of flexibility. This means forgoing the ability to specify arbitrary handling of each packet and seeking a more limited, but still useful, range of actions.

An OpenFlow Switch consists of at least three parts:

- A Flow Table, with an action associated with each flow entry, to tell the switch how to process the flow.
- A Secure Channel that connects the switch to a remote control process (called the controller), allowing commands and packets to be sent between a controller and the switch using.
- The OpenFlow Protocol, which provides an open and standard way for a controller to communicate with a switch. By specifying a standard interface (the OpenFlow Protocol) through which entries in the Flow Table can be defined externally, the OpenFlow Switch avoids the need for researchers to program the switch.

A dedicated OpenFlow Switch is a dumb data-path element that forwards packets between ports, as defined by a remote control process. Figure 2.4 shows an example of an OpenFlow Switch. In this context, flows are broadly defined, and are limited only by the capabilities of the particular implementation of the Flow Table. For example, a flow could be a TCP connection, or all packets from a particular MAC address or IP address, or all packets with the same VLAN tag, or all packets from the same switch port. For experiments involving non-IPv4 packets, a flow could be defined as all packets matching a specific (but non-standard) header. Each flow entry is associated with zero or more actions that dictate how the switch handles matching packets. If no forward actions are present, the packet is dropped. Action lists for inserted flow entries must be processed in the order specified. However, there is no packet output ordering guaranteed within a port. The three basic actions associated with each flow entry are:

- Forward this flows' packets to a given port (or ports). This allows packets to be routed through the network. In most switches this is expected to take place at line rate.
- Encapsulate and forward this flows' packets to a controller. Packet is delivered to Secure Channel, where it is encapsulated and sent to a controller. Typically used for the first packet in a new flow, so a controller can decide if the flow should be added to the Flow Table. Moreover, in some experiments, it could be used to forward all packets to a controller for processing.
- Drop this flows' packets. Can be used for security, to curb denial of service attacks, or to reduce spurious broadcast discovery traffic from end-hosts.

CHAPTER 3

Solution Proposed

3.1 Network-Based Mobility

First, we are going to present the network-based solution developed to manage IPv6 mobility issues through the SDN paradigm. This solution is completely transparent to the user and allows to multiple mobile nodes to attach and perform different types of handover processes, intra- and inter-district.
For a better understanding of the solutions proposed, we are going to highlight some main concepts of the CROWD architecture. This kind of architecture is based on the concept of district which is a single technology geographical region controlled by a SDN controller, The CROWD Local Controller (CLC). Optimizations on the management of the districts are done in an upper level controller, The CROWD Regional Controller, which has a global scope of the architecture. Once the concepts of district, CLC and CRC are clearer we are going to focus on the different interactions which can be performed by a mobile node within the CROWD architecture assuming the network-based approach.

3.1.1 Attachment

Figure 3.1, presents the initial connection of a terminal to a CROWD district. In this case, we are assuming the district is composed of IEEE 802.11 Access Points, an OpenFlow capable backhaul connecting the APs to the DMM-GWs and a CLC. Upon attachment of the Mobile Node, the AP generates an LLC message that serves as mechanism to update the forwarding table in the switches connecting the AP to the gateway. In this case, as all the network is OpenFlow capable, this LLC message is encapsulated in an OpenFlow message and sent to CLC D1. The LLC message contains the MAC address of the terminal and the MAC address of the AP, hence the CLC can use this message to trigger the detection of mobility or to trigger the mechanisms required for a node to join the network. In the case depicted in Fig. 3.1, CLC1 does not have any previous entry of the terminal on its BC, hence it assumes a new terminal has attached to the network. Then, the CLC assigns an IPv6 prefix and a DMM-GW to be used by the terminal and stores this information on its BC.

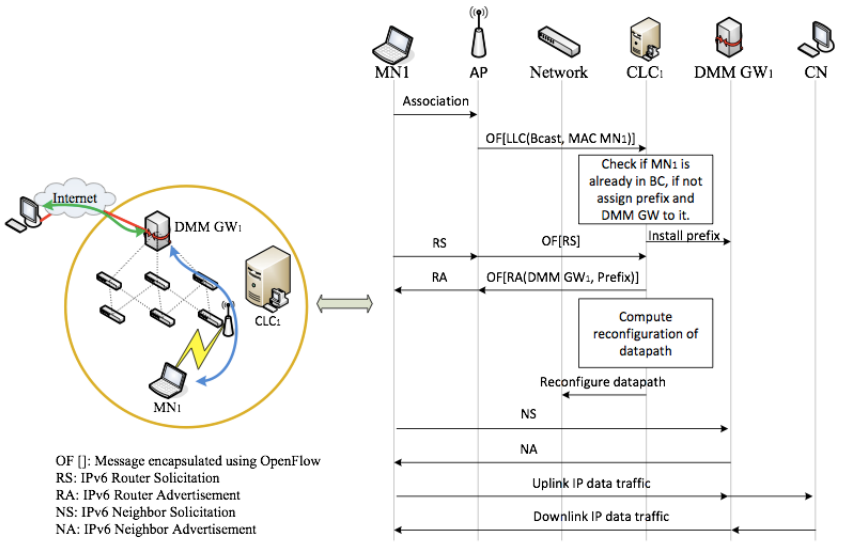


Figure 3.1: Network-based mobility

The standard procedure followed by a terminal after successful attachment to a new network includes the sending of a RS, in order to configure its IP address using IPv6 Stateless auto-configuration (SLAC). As in the case of the LLC message, the network encapsulates the RS message and sends it to the CLC. The CLC uses an RA message to answer the RS, providing the prefix and default router (DMM GW1) selected before. Hence, through the mediation of the CLC, highjacking the RA functionality of the network, we are able to control the IP level attachment of the terminal within the CROWD network. At this point in time, the CLC is able to compute the required match rules and data path modifications required to forward the terminals' packets to the selected DMM-GW. These modifications are configured into the network through the OpenFlow protocol, requiring several message exchanges among CLC1 and the different switches conforming the path between the terminal and DMM GW1. Once the data-path is configured, packets originated at the terminal with layer 2 destination the DMM-GW are transparently forwarded at layer 2. This behavior is completely transparent to the layer 3 stack of the terminal, which sees the path terminal until DMM-GW as a single hop. Finally, the terminal after performing a Neighbor Discovery procedure, is able to exchange packets with any CN through DMM GW1.

3.1.2 Intra-district

Figure 3.1, presents the initial connection of a terminal to a CROWD district. In this case, we are assuming the district is composed of IEEE 802.11 Access Points, an OpenFlow capable backhaul connecting the APs to the DMM-GWs and a CLC. Upon attachment of the terminal (MN1 in Fig. 3.1), the AP generates an LLC message1 that serves as mechanism to update the forwarding table in the switches connecting the AP to the gateway. In this case, as all the network is OpenFlow capable, this LLC message is encapsulated in an OpenFlow message and sent to CLC1. The LLC message contains the MAC address of the terminal and the MAC address of the AP, hence the CLC can use this message to trigger the detection of mobility or to trigger the mechanisms required for a node to join the network. In the case depicted in Fig.3.1, CLC1 does not

have any previous entry of the terminal on its BC, hence it assumes a new terminal has attached to the network. Then, the CLC assigns an IPv6 prefix and a DMM-GW to be used by the terminal and stores this information on its BC. The standard procedure followed by a terminal after successful attachment to a new network includes the sending of a RS, in order to configure its IP address using IPv6 Stateless auto-configuration (SLAC). As in the case of the LLC message, the network encapsulates the RS message and sends it to the CLC. The CLC uses an RA message to answer the RS, providing the prefix and default router (DMM GW1) selected before. Hence, through the mediation of the CLC, highjacking the RA functionality of the network, we are able to control the IP level attachment of the terminal within the CROWD network. At this point in time, the CLC is able to compute the required match rules and data path modifications required to forward the terminals' packets to the selected DMM-GW. These modifications are configured into the network through the OpenFlow protocol, requiring several message exchanges among CLC1 and the different switches conforming the path between the terminal and DMM GW1. Once the data path is configured, packets originated at the terminal with layer 2 destination the DMM-GW are transparently forwarded at layer 2. This behavior is completely transparent to the layer 3 stack of the terminal, which sees the path terminal – DMM-GW as a single hop. Finally, the terminal, after performing a Neighbor Discovery procedure [11], is able to exchange packets with any CN through DMM GW1.

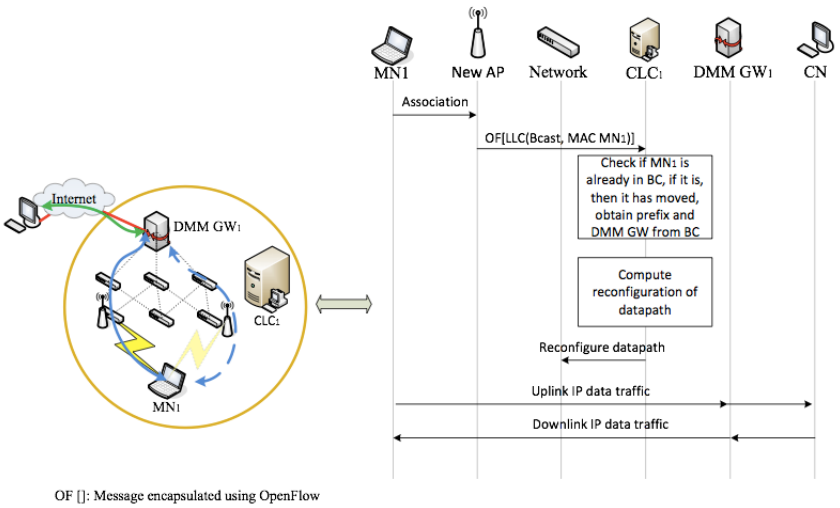


Figure 3.2: Intra-district handover

The case of handover is illustrated in Fig. 3.2, where the terminal attaches to a second AP within the same district. For the case of mobility within a district, the only required change is to modify the data-path, so packets are forwarded between the new AP and the DMM-GW assigned to the terminal. The process followed corresponds to the flow diagram depicted in Fig. 4.3. Upon attachment to the new AP, the AP generates an LLC message including the MAC address of the terminal and the MAC address of the AP it has connected to. Upon reception of this message, the CLC is able to identify that the terminal has moved, since it stores on its BC the MAC address of the AP the terminal was connected to (note this is an example of ID, and any other

terminal ID could be used). Once this information is obtained, the CLC is able to compute the required modifications to the data-path used by the terminal's packets, in such a way that, after modifying accordingly the OpenFlow configuration on the backhaul, the packets will flow from the new AP to the old DMM-GW (DMM GW1).

3.1.3 Inter-district

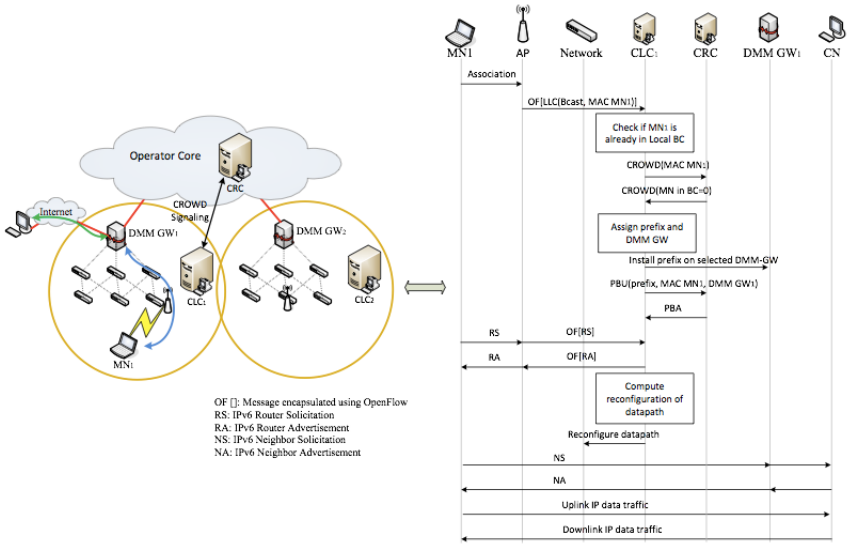


Figure 3.3: Inter-district attachment

The CROWD solution for inter-district mobility works on top of the intra-district mobility solution, and relies on two main entities, the CLC and the CRC. Since the solution works on layer 3, it is required the definition of new APIs to control the IP layer configuration of the DMM-GW. Specifically, we need two new APIs, one to convey information about the node, such as the IPv6 prefix and DMM-GW assigned to it and a second one to configure the IP layer of the DMM-GW the prefixes reachable through the interfaces and to setup an IP in IP tunnel. Fig. 3.3, shows the initial attachment phase of the CROWD inter-district mobility solution. The attachment to the network begins with the terminal associating to a point of attachment belonging to the district. As in the case of the intra-district mobility, this event triggers an LLC frame which is encapsulated in an OpenFlow message and forwarded to CLC1 . Through this message, the CLC is able to check on its local BC if the node is already attached to the district. If it is not the case, then it will contact the CRC, in order to check if the node is already registered in a previous district, and inter-district mobility is required. In the example depicted in Fig. 3.3, the terminal has not been attached previously to any CROWD network, so CLC1 is free to assign it any of the available IPv6 prefixes in the district. Once the CLC has decided the prefix and DMM-GW to be assigned to the terminal, it proceeds to install the prefix in the DMM-GW (DMM GW1). In this way, the CLC has an extra degree of flexibility, being able to assign arbitrarily the prefix to the selected gateway2 . In addition, the prefix, DMM-GW and terminal identification (MAC address) are notified to the CRC, which is able to keep track of the previous

attachments of the terminal. Once this process ends, the rest of the procedure is exactly the same as the one performed for the intra-district mobility, depicted in Fig. 3.1.

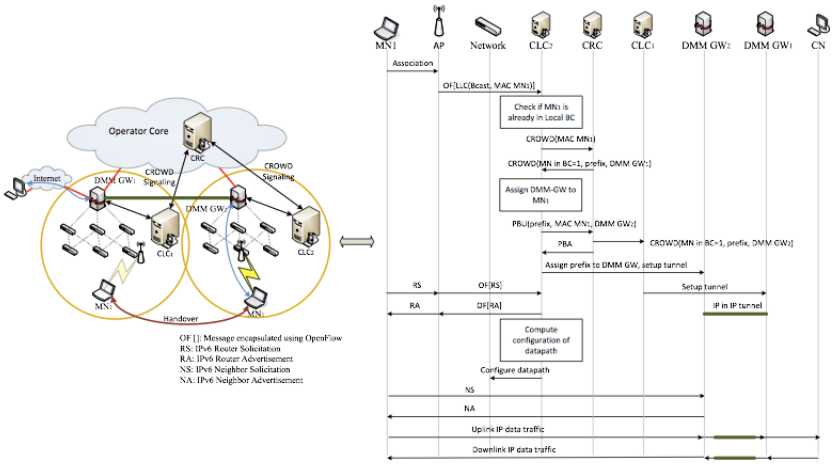


Figure 3.4: Inter-district handover

Fig. 3.4, presents the procedure of a handover between two different CROWD districts. The procedure assumes that the initial attachment process has been carried on as presented above. The procedure relies on the communication among the CLCs being orchestrated by the CRC. Basically, the CRC behaves as a data-base containing the list of previous DMM-GWs to be considered while performing handover. The configuration of the IP layer on the DMM-GWs and the tunnel setup among them is handled locally by the CLCs on each district. The procedure starts by the terminal attaching to an AP in a different district. As in previous cases, this event triggers the CLC (CLC2) to check if the node is registered on its internal BC. As this is the first time the terminal attaches to the district, the CLC asks the CRC for previous registrations. In this case, the CRC has information regarding the terminal, informing the CLC of the prior connection of the terminal to DMM GW1 and the prefix used. With this information, CLC2 is able to decide the DMM-GW (DMM GW2) to be used within this district and provides the CRC with this information. The CRC store this information on its local BC for future reference. At this point, several procedures are performed in parallel. First, the CRC informs CLC1 of the new location of the MN; with this information, CLC1 can configure DMM GW1 with an IP in IP tunnel connection with DMM GW2 and change the routes at DMM GW1 so the prefix used by the terminal is routed through the tunnel. In parallel, CLC2 configures the new prefix in DMM GW2 and sets up the IP in IP tunnel towards DMM GW1. Once the tunnel is established, the configuration of the data path in the new network is performed as in previous cases. When all the procedure is complete, packets between the CN and M N1 are forwarded first to DMM GW1, which tunnels them to DMM GW2. After DMM GW2 , the OpenFlow configured data path takes care of forwarding the packets to the appropriate location of MN1 within the district.

3.2 Host-Based Mobility

This section presents the global mobility host-based solution The idea behind this solution is to enable the terminal to maintain connectivity when it has roamed to a non-CROWD. Figure 3.5 presents the proposed solution adopted by CROWD: As the network-based proposals presented in above sections, this solution relies on the split of control and data planes, reusing concepts from traditional IP mobility solutions. In this case, we have chosen Mobile IPv6 as the basis of the proposal. As opposed to the diagram, the procedure starts with MN roaming to a point of attachment belonging to a CROWD network. Upon attachment, the terminal performs standard IPv6 procedures, soliciting a prefix or a DHCP lease. Once the paths are installed on every OpenFlow entity of the current district and after the device has acquired an IPv6 address, it can start generating traffic within its district. Once the Mobile Node has completed all Network-Based steps, it can attach to a non CROWD network. When this happens, in order to regain IP connectivity, the mobile node sends a message to the CROWD Regional Controller announcing the handover inter-network process. The terminal uses a MIPv6 BU message, which contains the newly obtained IPv6 address as CoA. This message is sent to the CRC, that in this case acts as HA for the terminal. Upon reception of the message, the CRC contacts the CLCs in charge of managing this terminal, e.g., the CLCs that control the prefix the terminal is using as HA. Once the CLC is aware of the terminal needs, it commands the previous DMM-GW serving the terminal, to create an IPv6 in IPv4 tunnel with destination the terminals' new IPv6 address. Through this solution, packets sent by the CN arrive at the DMM-GW that was previously serving MN (DMM GW1 in Fig. 3.5), to be forwarded through the tunnel to the terminal. The same behavior is enforced for uplink packets.

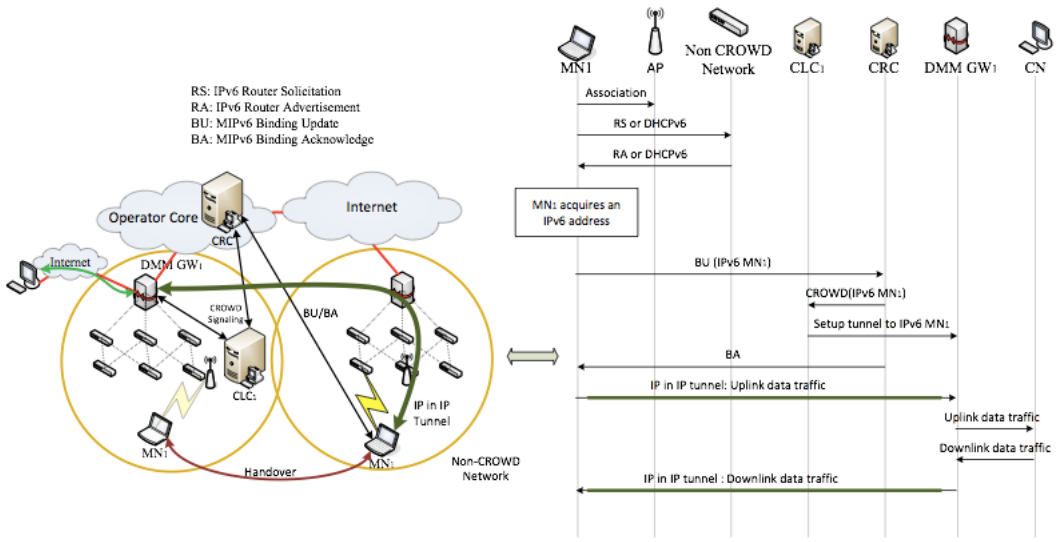


Figure 3.5: Host-Based mobility

3.3 Hardware

The next section will display all the hardware elements which are used to develop the solution proposed.

- **CROWD Regional Controller (CRC)**: Top level controller. Manages mobility inter-district on its regional network. To do that,it has a cache containing tuples of: Mobile Nodes attached to the network – DMM Gateway of the district they are attached to. It also has a cache of tunnels created during all the inter-districts handover. For this element we have chosen to use a Linux computer (Fig.3.6) running Python 2.7.3 with the following specs: OS: Debian 7.4 Wheezy - CPU: Intel(R) Pentium(R) 4 CPU 2.80GHz - RAM: 1 GB.
- **CROWD Local Controller (CLC)**: Middle level controller. Manages its local district. For this element we have chosen to use another computer running Ubuntu for each one of the two CLC of the test bed and the Ryu OpenFlow controller. We also have chosen to use a Linux computer (Fig.3.6) running Python 2.7.3 with the following specs: OS: Debian 7.4 Wheezy - CPU: Intel(R) Pentium(R) D CPU 3.20GHz - RAM: 1 GB.

Figure 3.6: CLCs and CRC

Figure 3.7: Mobile Node

- **Mobile Node**: It is the host of the OpenFlow network. It does not have any kind of OpenFlow knowledge; its only requirement is to have wireless connection and an IPv6 neighbor discovery (ICMPv6 [12]) protocol running. For this element we have chosen to use a laptop computer with Linux (Fig.3.7) running *Python 2.7.5* with the following specs: *OS: Ubuntu 13.10 Saucy - CPU: Intel(R) Core(TM) i5-4300 CPU @ 1.90GHz - RAM: 8 GB.*

- **DMM Gateway**: It is a non OpenFlow device. It is a normal router providing the OpenFlow network of Internet connection. It is linked to the Local Controller to be remotely configured. There is one per district, and it will be an Alix PC [13] device (Fig.3.8) running *Ubuntu 12.04.4 LTS - CPU: AMD Geode LX800 500 MHz - RAM: 250 MB.*

- **Access Point**: An OpenFlow switch with the traditional behavior of a wireless access point. Provides connection to the Mobile Node. We will use two Alix PC as access point per each district and they both will be an Alix PC [13] device (Fig.3.8) running *Ubuntu 12.04.4 LTS - CPU: AMD Geode LX800 500 MHz - RAM: 250 MB with Open vSwitch.*

- **Node**: An OpenFlow switch with its wireless connection disabled. It has the same behavior of the Access points, with the only difference of the wireless connection. When a Node is directly connected to a Gateway/DMM Gateway it is considered as an OpenFlow Edge node. There is one per district, and it will be a Linksys WRT54GL [14] (Fig.3.9) running OpenFlow Pantou.

Figure 3.8: Alix

Figure 3.9: Liknsys WRT54GL

3.4 Software

The next section will display all the software elements which are used to develop the solution proposed.

- **Scappy**[15]: Scappy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols
- **Ryu**[16]: Ryu is a component-based software defined networking framework. Ryu provides software components with well defined API that make it easy for developers to create new network management and control applications. Ryu supports various protocols for managing network devices, such as OpenFlow.
- **Open vSwitch**[17]: Open vSwitch is a production quality, multilayer virtual switch licensed under the open source Apache 2.0 license. It is designed to enable massive network automation through programmatic extension, while still supporting standard management interfaces and protocols.
- **Pantou**[18]: Pantou turns a commercial wireless router/Access Point to an OpenFlow-enabled switch. OpenFlow is implemented as an application on top of OpenWrt. Pantou is based on the BackFire OpenWrt release (Linux 2.6.32). The OpenFlow module is based on the Stanford reference implementation (user space).

CHAPTER 4

CROWD Connectivity Management

4.1 Introduction

The Connectivity Management prototype has been developed over a SDN framework specially suited for the fast development of applications using the OpenFlow protocol to control data path elements. This framework is the Ryu (http://osrg.github.io/ryu/) framework, which allows the development of application using OpenFlow versions from 1.0 to 1.4 using Python. In order to understand the different challenges we have faced while developing this application, it is worth highlighting the most important concepts behind the architecture of the Ryu controller. The Ryu controller is a software component providing specific APIs for the use of OF protocol to develop control applications. An application using the Ryu framework requires to be written in a special way. Ryu applications are event based, this means that the applications will be triggered based on a specific event occurrence such as the reception of an OF packet, and the developer needs to write specific functions to handle the different possible events consumed by the application. Although the concept of event based programming is not new and facilitates the use of Ryu to write simple applications, it has a major caveat; Any functionality that the application needs to implement, which does not relay on simple reaction upon an event triggering, must be implemented as a separated thread with all the synchronization issues imposed by this way of doing things.

In addition to the API for the OF protocol, the Ryu controller also includes some classes which can be imported and inherited in order to write our own application. One of the benefit of such approach is that some of the default events that the application must support are already caught and tackled in the classes provided, hence inheriting from these classes solves the need of writing specific code to handle default events.

In the case of the Connectivity Management application we have inherited from several classes, which consume the following events (this list is taken from the output of the application):

```
...
BRICK dpset
 CONSUMES EventOFPStateChange
 CONSUMES EventOFPSwitchFeatures
 CONSUMES EventOFPPortStatus
BRICK SimpleSwitch
 CONSUMES EventOFPPacketIn
```

```
 CONSUMES EventOFPEchoRequest
 CONSUMES EventOFPPortStatus
BRICK ofp_event
 PROVIDES EventOFPEchoRequest TO {'SimpleSwitch': set(['main'])}
 PROVIDES EventOFPStateChange TO {'dpset': set(['main', 'dead'])}
 PROVIDES EventOFPPacketIn TO {'SimpleSwitch': set(['main'])}
 PROVIDES EventOFPSwitchFeatures TO {'dpset': set(['config'])}
 PROVIDES EventOFPPortStatus TO {'dpset': set(['main']), 'SimpleSwitch': set(['main'])}
 CONSUMES EventOFPPortDescStatsReply
 CONSUMES EventOFPErrorMsg
 CONSUMES EventOFPSwitchFeatures
 CONSUMES EventOFPEchoRequest
 CONSUMES EventOFPHello
..
```

Each of the BRICK blocks in the above console output corresponds to a certain class and set of events consumed by the class. In the case of our application, the first BRICK used corresponds to the dpset class. This class name, dpset, means Data-path set and it is used to store a reference to all the data path elements (switches) building the controlled network. As the main objective of this class is to handle the connection of the different data path elements to the application, it consumes by default the events in charge of providing information on the features of the switches (EventOFPSwitch-Features), changes in the state of a data path element such as a change in its connection status to the controller (EventOFPStateChange) and changes in the port status such as the joining of a port to the set of controlled ports of a data path element (EventOFP-PortStatus). The second BRICK that appears in the console output corresponds to the SimpleSwitch class. This class corresponds to the template we have used for building the Connectivity Management application. As can be seen from the events consumed by the SimpleSwitch BRICK, our application basically reacts upon three events, the EventOFPPacketIn, EventOFPEchoRequest and EventOFPPortStatus. From these three events, the only one we have needed to code is the first one, EventOFPPacketIn. This event is triggered every time a user plane packet is forwarded by a data path element to the controller. The standard behavior of an OF switch when it encounters a packet not matching any of the rules configured in the matching table, is to encapsulate and forward it to the controller. Once the Ryu framework receives this packet, it forwards it to the applications consuming this event and triggers the function defined to handle it. The major part of the code implementing the Connectivity Management, is implemented in the function tackling this event, since the mobility and establishment of paths is performed based on the traffic received from the data path elements. The events EventOFPEchoRequest and EventOFPPortStatus are implemented just to handle error conditions. The EventOFPEchoRequest is used to handle the heartbeat message used to keep the control connection between the data path elements and the controller open. The Event OFPPortStatus is used to handle the changes in port status, overwriting the default behavior provided in the dpset class. The reason for implementing both event handlers is just to be able to debug in a better way the different error messages that the application may drop. Finally, the ofp_event BRICK corresponds to the main Ryu functionality that must be imported into every application using the Ryu API. It is basically a module in charge of orchestrating the consumption of events from the different classes. The functionality provided by this BRICK is similar to an event hub. On the one hand, it registers as the main consumer of the events solicited by the rest of BRICKs, while providing them to the rest of modules upon reception.

Once the fundamental concepts behind the development of the Connectivity Man-

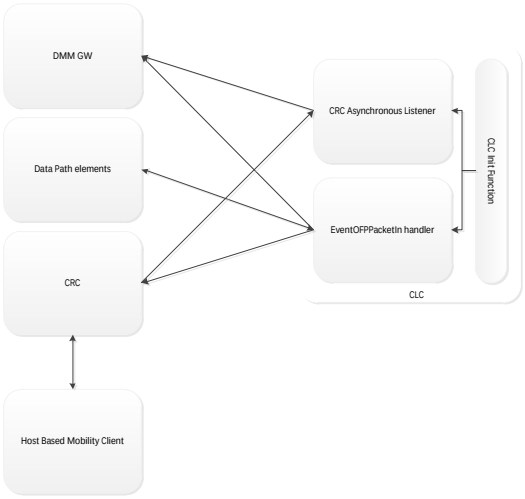


Figure 4.1: Architecture of the Connectivity Management implementation

agement application are explained, Figure 4.1 shows the different modules and the communication between them. The Connectivity Management application contains modules in five different entities:

- CROWD Local Controller: The Connectivity Management component within the CLC is in charge of managing the mobility operations within the district. The component at the CLC is divided in two separated modules, the EventOFPPacketIn handler and the CRC Asynchronous Listener. The former is in charge of configuring the data path elements based on the different packets forwarded to the controller and it also configures the layer-3 forwarding mechanisms at the DMM GW, such as setting up tunnels, installing routes etc.. The later corresponds to the thread continuously listening for indications coming from the CRC. This second module is specially important for the inter-district handover, since the CRC will notify the CLC when an MN connected to the district controlled by this CLC moves to another district. In this case, the CRC Asynchronous Listener must configure the IP tunnel in the DMM GW and setup the new layer 3 forwarding so packets addressed to the MN are forwarded to its new location (to the DMM GW serving the new district).

- CROWD Regional Controller: The Connectivity Management component within the CRC is in charge of keeping track of the current location of the MN while it is roaming across multiple districts. Every time an MN attaches to a district, the CLC EventOFPPacketIn handler informs the CRC in order to check if inter-district mobility mechanisms are needed. In case the CRC detects it is required to start the inter-district handover mechanism, it will contact the CRC Asynchronous Listener in the CLC previously serving the MN, so it can start the tunnel building process. Finally, the last functionality provided by the CRC is to behave as the central contact point for the Host-Based Mobility solution. When an MN attaches to a non-CROWD network, it can contact the CRC controlling the previous district where the terminal was attached, and request that the packets belonging to its open connections are forwarded to its new location through

an IPv6 in IPv4 tunnel. This process is similar in nature to an inter-district handover and the communication between the CLC and CRC is performed in similar ways.

- Data Path Elements: The different SDN controlled data path elements are managed by the CLC specifically for each district. In this way, the unique point of control in a district is the CLC, which issues OpenFlow 1.0 commands to the different data path elements. It is important to highlight that the CRC does not interact in any way with the data path elements, performing all communication directly with the CLC in charge of the district.

- DMM GW: For our implementation to work, we need mechanisms to control and configure IP related features of the DMM GW. In order to do so, we have created a network daemon running in the DMM GWs which is able to receive commands from the CLC and execute them. The set of commands implemented is rather simple and small, supporting the creation of IPv6 in IPv6 tunnels, IPv6 in IPv4 tunnels, set up routes and install configurations for source routing. The interface currently in use is based on an adhoc protocol created ex-proceso but we aim at extending the I2RS set of specifications to use them for this purpose.

- Host-Based Mobility Client: One of the requirements on the mobility solution provided by CROWD was its transparency for the end user. For the network based mobility solutions relying on DMM, this requirements is met, since the terminal does not require any extra protocol or modification to benefit from the mobility solution. On the contrary case, when the terminal moves to a network which is not CROWD enabled, it cannot benefit from the DMM solution and requires extra mechanisms to handle its mobility. As the target network is not a CROWD network, we cannot relay on any network-based mechanism to inform the CRC of the mobility of the terminal. Therefore, we need some improvements in the protocol stack of the terminal to be able to contact the CRC and inform it of its new location. This specific message exchange is the functionality included in this module, that needs to be installed in each terminal supporting mobility to non CROWD networks.

4.2 Testbed description

Figure 4.2 presents the schematic representation of the testbed deployed in order to test the DMM functionality. The testbed is composed of two CROWD WiFi districts and a non-CROWD IPv4-only access network. Although the different district and control elements are connected through a local network, we have tested and validated the behavior of the solution while communicating with nodes outside the local network. Both WiFi districts are identical in terms of software and hardware. Each district is composed by two OF enabled access points, an OF controlled switch and a non-OF enabled router playing the role of DMM GW. Each of the elements is connected to a central switch used for control purposes (blue lines in Figure 4.2) which also connects the district to the CROWD Local Controller for each district. The hardware used for each element is detailed in Section 3.3. Regarding the software used for the OF implementation in the data path elements, at the moment we use two different

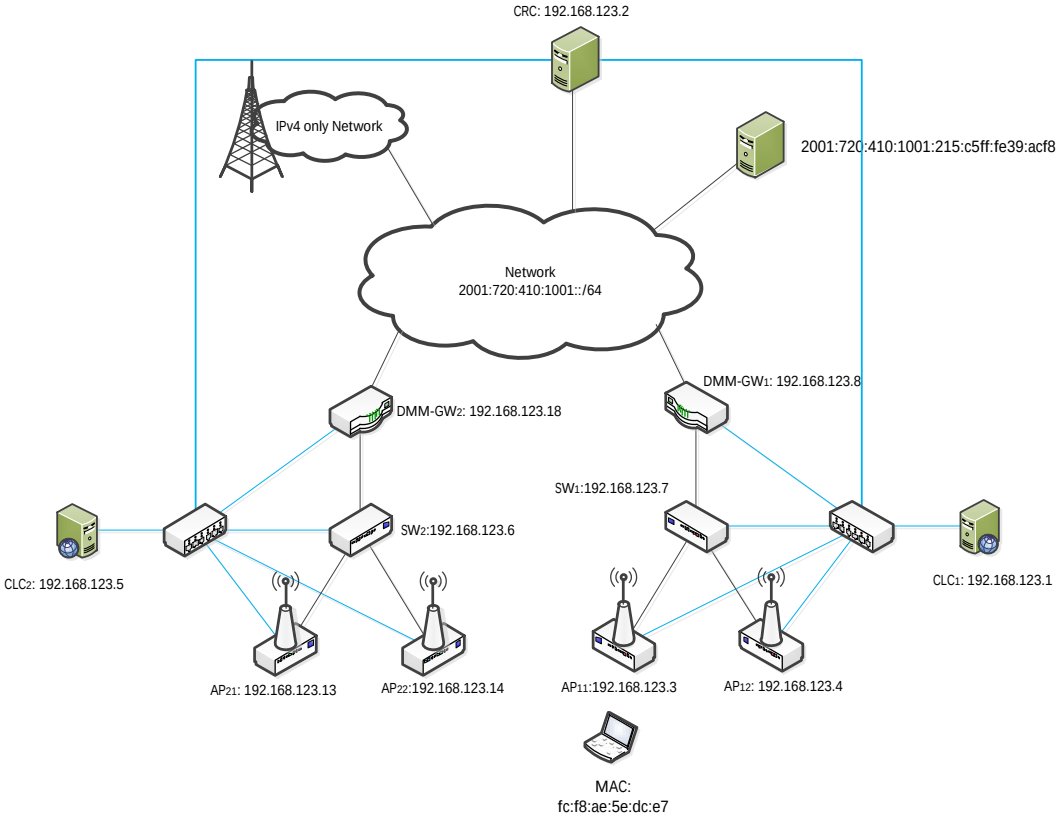


Figure 4.2: Testbed diagram

implementations. The first one corresponds to the Pantou[1] implementation of OpenFlow 1.0 for OpenWRT devices. The second one is the well known implementation of OpenFlow currently available for the Open Virtual Switch[2]. Although we are using both implementations due to different hardware constraints (the number of devices supporting Open vSwitch available at our lab is limited compared with the number of OpenWRT devices), we have already benchmarked each implementation and the results show a great advantage in terms of performance of using Open vSwitch compared with Pantou. Therefore, we aim at porting all our testbed to Alix devices in the near future.

4.2.1 Functional description

This section is devoted to the explanation and analysis of the DMM state machine. In order to provide a clear and detailed view of the application behavior, we will provide in the following flow diagrams and console output showing each of the steps performed. The use cases analyzed in this section are the following:

- Connection of data path elements to the controller
- MN Attachment to a district.

[1] `http://archive.openflow.org/wk/index.php/Pantou_:_OpenFlow_1.0_for_OpenWRT`
[2] http://openvswitch.org/

- Intra-district Handover.
- Inter-district Handover.
- Host-based Handover.

We start by analyzing the connection of the data path elements to the controller in next section.

4.2.2 Connection of the data path elements to the controller

Before any DMM functionality starts running, each of the data path elements must connect and create a secure TLS connection with the controller. We have decided to implement the control channel using an out-bound connection (blue line in Figure 4.2). The connection of all data path elements to the controller requires a certain amount of time and grows linearly with the number of data path elements, since the controller on its current implementation is not able to handle data path element connections in a concurrent way.

Listing 4.1 shows the output console at the controller while the different data path elements are connected for the district 1 in Figure 4.2.

Listing 4.1: Data path elements association to the controller

```
loading app CLC.py
District D1
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK dpset
  CONSUMES EventOFPStateChange
  CONSUMES EventOFPSwitchFeatures
  CONSUMES EventOFPPortStatus
BRICK SimpleSwitch
  CONSUMES EventOFPPacketIn
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPPortStatus
BRICK ofp_event
  PROVIDES EventOFPEchoRequest TO {'SimpleSwitch': set(['main'])}
  PROVIDES EventOFPStateChange TO {'dpset': set(['main', 'dead'])}
  PROVIDES EventOFPPacketIn TO {'SimpleSwitch': set(['main'])}
  PROVIDES EventOFPSwitchFeatures TO {'dpset': set(['config'])}
  PROVIDES EventOFPPortStatus TO {'dpset': set(['main']), 'SimpleSwitch':
      set(['main'])}
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPErrorMsg
  CONSUMES EventOFPSwitchFeatures
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPHello
RECTIFIER
bind on 192.168.123.1 port 9991

connected socket:<eventlet.greenio.GreenSocket object at 0xb13c6cc>
    address:('192.168.123.4', 41714)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0xb13c92c>
move onto config mode
EVENT ofp_event->dpset EventOFPSwitchFeatures
switch features ev version: 0x1 msg_type 0x6 xid 0x4f078578
    OFPSwitchFeatures(actions=4095,capabilities=199,datapath_id=58941412932L,
```

```
n_buffers=256,n_tables=255,ports={1:OFPPhyPort(port_no=1,hw_addr='00:c0:ca:1f:40:fc',
name='wlan0',config=0,state=0,curr=0,advertised=0,supported=0,peer=0),
2:OFPPhyPort(port_no=2,hw_addr='00:0d:b9:2e:96:46',name='eth2',config=0,state=0,
curr=520,advertised=1679,supported=655,peer=0), 3:
    OFPPhyPort(port_no=3,hw_addr='00:0d:b9:2e:96:44',name='eth0',config=0,state=1,
curr=513,advertised=1679,supported=655,peer=0), 65534:
    OFPPhyPort(port_no=65534,hw_addr='00:0d:b9:2e:96:44',name='of-switch',
config=1,state=1,curr=0,
advertised=0,supported=0,peer=0)})
move onto main mode
EVENT ofp_event->dpset EventOFPStateChange
DPSET: register datapath <ryu.controller.controller.Datapath object at 0xb13c2ac>

connected socket:<eventlet.greenio.GreenSocket object at 0xb13c76c>
    address:('192.168.123.3', 50804)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0xb13cf2c>
move onto config mode
EVENT ofp_event->dpset EventOFPSwitchFeatures
switch features ev version: 0x1 msg_type 0x6 xid 0x49da7e16
    OFPSwitchFeatures(actions=4095,capabilities=199,datapath_id=58941326156L,
n_buffers=256,n_tables=255,ports={1:OFPPhyPort(port_no=1,hw_addr='00:c0:ca:1f:6e:fe',
name='wlan0',config=0,state=0,curr=0,advertised=0,supported=0,peer=0),
2:OFPPhyPort(port_no=2,hw_addr='00:0d:b9:2d:43:4e',name='eth2',config=0,state=1,
curr=513,advertised=1679,supported=655,peer=0),
 3:OFPPhyPort(port_no=3,hw_addr='00:0d:b9:2d:43:4c',name='eth0',config=0,state=0,
 curr=520,advertised=1679,supported=655,peer=0),
     65534:OFPPhyPort(port_no=65534,hw_addr='00:0d:b9:2d:43:4c',name='of-switch',
 config=1,state=1,curr=0,advertised=0,supported=0,peer=0)})
move onto main mode
EVENT ofp_event->dpset EventOFPStateChange
DPSET: register datapath <ryu.controller.controller.Datapath object at 0xb13cbcc>

connected socket:<eventlet.greenio.GreenSocket object at 0xb13cc2c>
    address:('192.168.123.7', 58668)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0xb141e0c>
move onto config mode
EVENT ofp_event->dpset EventOFPSwitchFeatures
switch features ev version: 0x1 msg_type 0x6 xid 0x5a916110
    OFPSwitchFeatures(actions=3839,capabilities=199,datapath_id=150872184132L,
n_buffers=256,n_tables=2,ports={1:OFPPhyPort(port_no=1,hw_addr='00:1c10:44:35:b9',
name='eth0.0',config=0,state=0,curr=648,advertised=3599,supported=527,peer=0),
 2: OFPPhyPort(port_no=2,hw_addr='00:1c:10:44:35:b9',name='eth0.1',config=0,state=0,
 curr=648,advertised=3599,supported=527,peer=0),
 3:OFPPhyPort(port_no=3,hw_addr='00:1c:10:44:35:b9',name='eth0.2',config=0,state=0,
 curr=648,advertised=3599,supported=527,peer=0),
  4:OFPPhyPort(port_no=4,hw_addr='00:1c:10:44:35:b9',name='eth0.3',config=0,state=0,
  curr=648,advertised=3599,supported=527,peer=0)})
move onto main mode
EVENT ofp_event->dpset EventOFPStateChange
DPSET: register datapath <ryu.controller.controller.Datapath object at 0xb13cf6c>
```

The output console presented in Listing 4.1 is divided in two main parts. The first one corresponds to the initialization of the core classes or BRICKS used by the application, as explained in Section 4.1. The second one corresponds to the connection between the different data path elements and the controller. As can be seen in the listing, in this case (see Figure 4.2 for reference) there are three OF enabled data path

elements. The connection of each of them can be identified by the connection of different sockets to the addresses 192.168.123.4, 192.168.123.3 and 192.168.123.7.

Lets focus on the connection of the data path element 192.168.123.3 ($AP_1$1 in Figure 4.1):

```
..........
connected socket:<eventlet.greenio.GreenSocket object at 0xb13c76c>
    address:('192.168.123.3', 50804)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0xb13cf2c>
move onto config mode
EVENT ofp_event->dpset EventOFPSwitchFeatures
switch features ev version: 0x1 msg_type 0x6 xid 0x49da7e16
    OFPSwitchFeatures(actions=4095,capabilities=199,datapath_id=58941326156L,
n_buffers=256,n_tables=255,ports={1:OFPPhyPort(port_no=1,hw_addr='00:c0:ca:1f:6e:fe',
name='wlan0',config=0,state=0,curr=0,advertised=0,supported=0,peer=0),
 2:OFPPhyPort(port_no=2,hw_addr='00:0d:b9:2d:43:4e',name='eth2',config=0,state=1,
 curr=513,advertised=1679,supported=655,peer=0),
 3:OFPPhyPort(port_no=3,hw_addr='00:0d:b9:2d:43:4c',name='eth0',config=0,state=0,
 curr=520,advertised=1679,supported=655,peer=0),
     65534:OFPPhyPort(port_no=65534,hw_addr='00:0d:b9:2d:43:4c',name='of-switch',
 config=1,state=1,curr=0,advertised=0,supported=0,peer=0)})
move onto main mode
EVENT ofp_event->dpset EventOFPStateChange
DPSET: register datapath <ryu.controller.controller.Datapath object at 0xb13cbcc>
..............
```

If we analyze the above console output, we can see that the amount of information provided by the data path element to the controller is quite detailed. Once Ryu detects the connection of a data path element, it starts triggering events to gather information from the data path element. This information includes the OFPSwitchFeatures event, which triggers several requests to the data path element, obtaining the different information about the capabilities of the ports associated to this data path element. It is interesting to highlight that in the case of the data path shown above, the first of the ports name is wlan0, which corresponds to the wireless interface of the AP. This port is recognized as an Ethernet port without properties. This is one of the key challenges of working with wireless interfaces and current openflow implementations; The OF controller does not know anything about the wireless nature of the interface.

4.2.3 MN attachment to a district.

In this section we present the functionality and steps performed by the CLC and CRC when a new MN joins the network. Figures 4.3 and 4.4 show the flow diagram of the DMM application both at the CLC and CRC respectively. Both figures present the complete flow diagram of the DMM operation for all possible cases, highlighting in gray the actual steps performed for this specific case. Both figures are used as reference while explaining the console output shown in Listing 4.2.

Listing 4.2: Attachment detection and RA generation

```
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Message received from AP switch with address: 192.168.123.3 in port 1
----------------------------------------------------------------
mac_src: fc:f8:ae:5e:dc:e7, mac_dst: ff:ff:ff:ff:ff:ff
----------------------------------------------------------------
ACL for Mobile Nodes is disabled.
Packet type is LLC_SNAP
Destination Mac: ff:ff:ff:ff:ff:ff
```

```
Mobile Node IS NOT in the BCE
Packet content
{'sent_time': 0, 'fields': {'dsap': 0, 'ssap': 1, 'ctrl': 175}, 'aliastypes':
    [<class 'scapy.layers.l2.LLC'>], 'post_transforms': [], 'underlayer': <Dot3
    dst=ff:ff:ff:ff:ff:ff src=fc:f8:ae:5e:dc:e7 len=6 |<LLC dsap=0x0 ssap=0x1
    ctrl=175 |<Raw load='\x81\x01\x00' |>>>, 'fieldtype': {'dsap': <Field
    (LLC).dsap>, 'ssap': <Field (LLC).ssap>, 'ctrl': <Field (LLC).ctrl>}, 'time':
    1401880549.994878, 'initialized': 1, 'overloaded_fields': {}, 'packetfields':
    [], 'payload': <Raw load='\x81\x01\x00' |>, 'default_fields': {'dsap': 0,
    'ssap': 0, 'ctrl': 0}}
    -->CRC doesn't know MN
IPv6: ICMP6 - Router Solicitation, source MAC address fc:f8:ae:5e:dc:e7, switch
    58941326156
Mobile Node (fc:f8:ae:5e:dc:e7) has performed a Router Solicitation
NEW MOBILE NODE CONNECTED TO THE SDN
Type: N Prefix: 00 mask: 0 defGW: 00
IPv6: ICMP6 - MAC fc:f8:ae:5e:dc:e7 added to BCE.
Mac mcast saved: 33:33:ff:5e:dc:e7
Prefix saved in the Pool: 2101::
IPv6: ICMP6 - Router advertisement prefix: 2101::
IPv6: ICMP6 - Sent RA back!
ip -6 route add 2101::/64 dev eth0
IPv6: ICMP6 - Communicating with CRC

DEBUG INFO:
------------------------------------------------------------
Received packet is LLC_SNAP
New MN:
 MN(fc:f8:ae:5e:dc:e7), AP(192.168.123.3), MN_mcast(33:33:ff:5e:dc:e7)
Route installed in GW with prefix assigned and the internal interface of the SDN
Notify CRC:
 MN(fc:f8:ae:5e:dc:e7), prefix(2101::), mask(64) y
     DG(2001:720:410:1001:20d:b9ff:fe2e:9855)
Router Advertisement sent back, MN should install prefix, mask y generate an IPv6
------------------------------------------------------------
============> _packet_in_handler: 0.0209469795227 seconds
```

Once an MN attaches to an AP (in this case the MN attaches to AP with control IP 192.168.123.3 in the wlan0 interface, port 1), the switch in the data path element generates an LLC message which is forwarded to the controller since the OF table in the data path element does not contain any entry dealing with LLC messages. This message is forwarded by the Ryu framework to the DMM application. The packet received can be seen in Listing 4.2, below the Packet content line, and is repeated here for clarity:

```
....
Packet content
{'sent_time': 0, 'fields': {'dsap': 0, 'ssap': 1, 'ctrl': 175}, 'aliastypes': [<class
    'scapy.layers.l2.LLC'>], 'post_transforms': [], 'underlayer': <Dot3
    dst=ff:ff:ff:ff:ff:ff src=fc:f8:ae:5e:dc:e7 len=6 |<LLC dsap=0x0 ssap=0x1 ctrl=175 |<Raw
    load='\x81\x01\x00' |>>>, 'fieldtype': {'dsap': <Field (LLC).dsap>, 'ssap': <Field
    (LLC).ssap>, 'ctrl': <Field (LLC).ctrl>}, 'time': 1401880549.994878, 'initialized': 1,
    'overloaded_fields': {}, 'packetfields': [], 'payload': <Raw load='\x81\x01\x00' |>,
    'default_fields': {'dsap': 0, 'ssap': 0, 'ctrl': 0}}
....
```

Following the flow diagram depicted in Figure 4.3, the controller will check if the node is locally new by checking the local BC. In this case the node is new in the

district, hence the CLC checks with the CRC if the node is new in the CROWD domain or mobility features must be put in place for some of the MNs' traffic. Figure 4.4 shows the CRC execution when contacted by a CLC. After receiving an advertisement message, the CRC will check on its local BC if the node has been previously registered at any district. In this case, the CRC will inform the CLC that the MN has not been previously seen in the CROWD domain.

Upon reception of the confirmation of new MN from the CRC, the CLC proceeds to assign the MN a prefix and a DMM GW. In this case, the prefix assigned corresponds to the 2101:: and the DMM GW is the 2001:720:410:1001:20d:b9ff: fe2e:9855 (this IPv6 address corresponds to the global IP address assigned to the $DMMGW_1$). After the CLC decides the prefix to delegate to the MN, it generates a Router Advertisement including the information on the prefix and default gateway, and forwards it encapsulated in an OF message to the MN. This is performed before the MN sends a Router Solicitation, in order to speed up the attachment of the MN to the network. In parallel to this, the CLC also contacts with the DMM GW and creates a route for the assigned prefix.

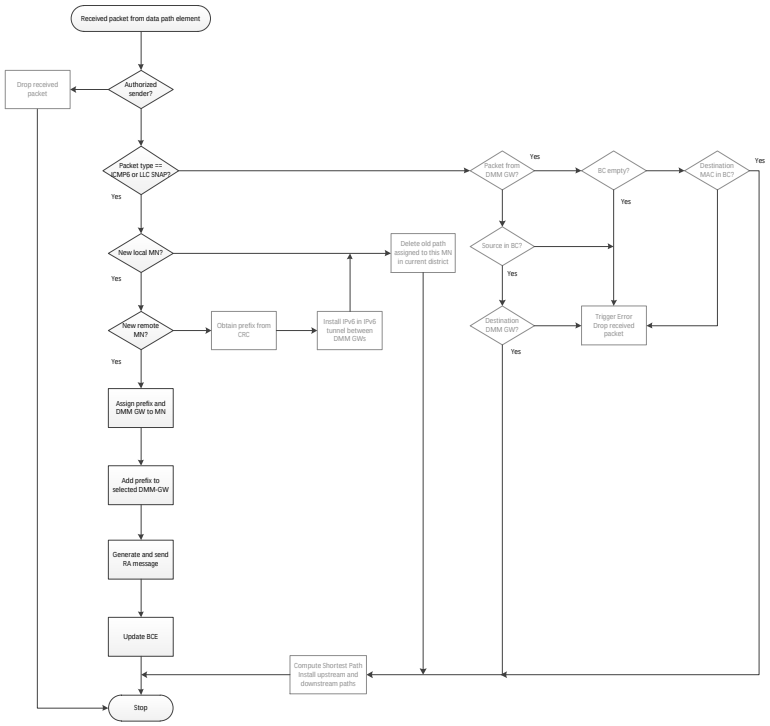


Figure 4.3: Initial attachment to a district, CLC

It is worth highlighting that at this point the MN has obtained a prefix and next hop information from the network and the CLC has registered the MN locally at the CLC and remotely at the CRC, but there is yet no path connecting the MN to the DMM GW. At this point, the CLC is still not able to setup any traffic match since it does not know what kind of traffic is going to be addressed at the MN, hence the CLC needs to wait until some upstream and downstream packet is received in order to install the paths. This operation is presented in Figure 4.5 and Listings 4.3 and 4.4, where the different steps performed by the CLC when a new packet is received are shown.

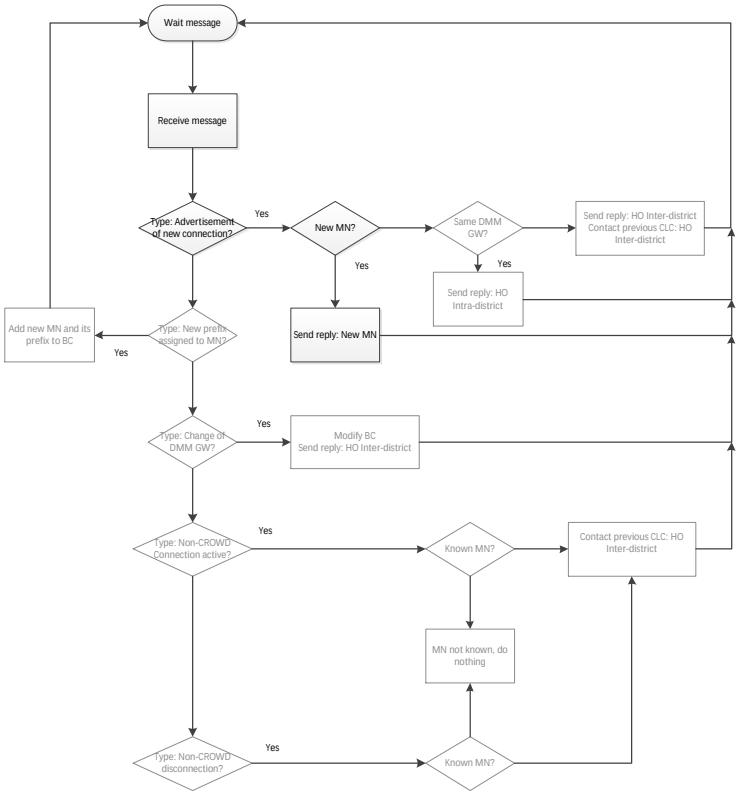


Figure 4.4: Initial attachment to a district, CRC

In this example, we only show the interception of an ICMPv6 (Echo Request) and the installation of the upstream data path. Note that this procedure is independent of the traffic kind and a similar approach must be performed when a downstream packet is received.

Listing 4.3: Packet interception

```
EVENT ofp_event->SimpleSwitch EventOFPPacketIn

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Message received from AP switch with address: 192.168.123.3 in port 1
------------------------------------------------------
mac_src: fc:f8:ae:5e:dc:e7, mac_dst: aa:bb:cc:dd:ee:ff
------------------------------------------------------
ACL for Mobile Nodes is disabled.
IPv6 Packet
BCE Content:
------------------------------------------------------
MAC: fc:f8:ae:5e:dc:e7 AP: 192.168.123.3 MAC_MCAST: 33:33:ff:5e:dc:e7 IPv6: 0000
------------------------------------------------------
{'sent_time': 0, 'fields': {'nh': 58, 'src': '2101::39fe:4b3:5e70:b8e6', 'dst':
    '2001:720:410:1001:215:c5ff:fe39:acf8', 'version': 6L, 'hlim': 64, 'plen': 64,
    'fl': 0L, 'tc': 0L}, 'aliastypes': [<class 'scapy.layers.inet6.IPv6'>],
    'post_transforms': [], 'underlayer': <Ether dst=aa:bb:cc:dd:ee:ff
    src=fc:f8:ae:5e:dc:e7 type=0x86dd |<IPv6 version=6L tc=0L fl=0L plen=64
    nh=ICMPv6 hlim=64 src=2101::39fe:4b3:5e70:b8e6
    dst=2001:720:410:1001:215:c5ff:fe39:acf8 |<ICMPv6EchoRequest type=Echo Request
```

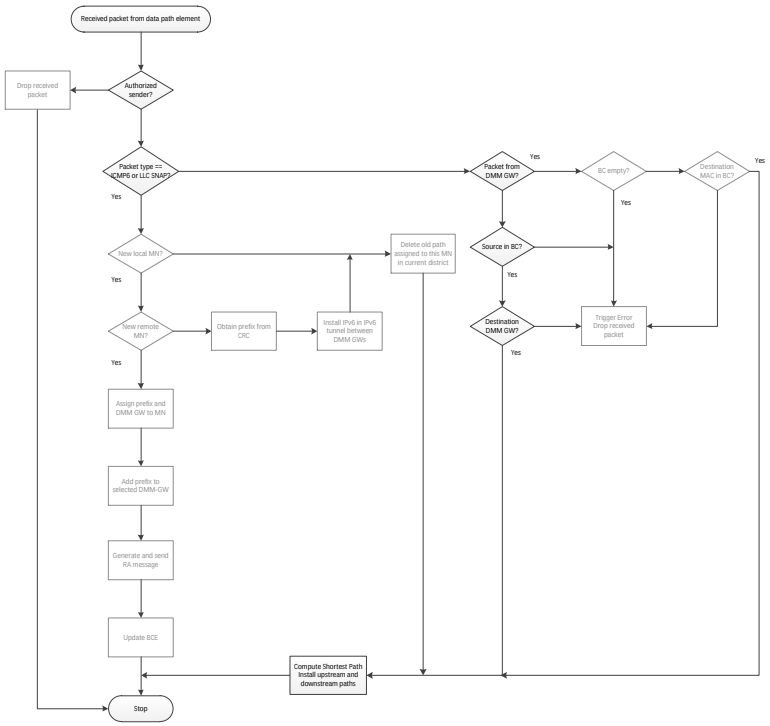


Figure 4.5: Packet interception and path setup

```
    code=0 cksum=0x472 id=0x54cc seq=0x1 data=hidden}

Mobile node IP obtained from the packet: 2101::39fe:4b3:5e70:b8e6

{'sent_time': 0, 'fields': {'src': 'fc:f8:ae:5e:dc:e7', 'dst': 'aa:bb:cc:dd:ee:ff',
    'type': 34525}, 'aliastypes': [<class 'scapy.layers.l2.Ether'>]}

Mobile node MAC obtained from the packet: fc:f8:ae:5e:dc:e7

Update multicast mac stored in BCE of the mobile node with mac fc:f8:ae:5e:dc:e7
Mac mcast obtenida: 33:33:ff:70:b8:e6
Update IPv6 in BCE of the mobile node
BCE Content:
-----------------------------------------------------
MAC: fc:f8:ae:5e:dc:e7 AP: 192.168.123.3 MAC_MCAST: 33:33:ff:70:b8:e6 IPv6:
    2101::39fe:4b3:5e70:b8e6
-----------------------------------------------------
```

The reception of a packet originated either by an MN or the DMM GW by the CLC means that there is no rule installed regarding the MN in the Access Point or in the DMM GW connection to the OF network. In this case, the packet is processed first at layer 3, in order to obtain the IPv6 address used and later at layer 2, in order to obtain the MAC address of the MN. Based on this information, the CLC is able to compute the different layer 2 addresses that must be setup as rules in the OF network, so the traffic coming from the MN reaches the DMM GW. In particular, the CLC computes the directed layer 2 multicast address used by IPv6 Neighbor Solicitation messages, the unicast layer 2 address and the layer 2 address corresponding to the IPv6 all hosts

address.

Based on this addresses, Listing 4.4 shows the installation of the three paths required to bring connectivity to the MN in the OF network.

Listing 4.4: Installation of upstream paths

```
BCE Content:
-----------------------------------------------------
MAC: fc:f8:ae:5e:dc:e7 AP: 192.168.123.3 MAC_MCAST: 33:33:ff:70:b8:e6 IPv6:
    2101::39fe:4b3:5e70:b8e6
-----------------------------------------------------
Installig upward path from MN to GW
IPv6: - Installing Path...
IPv6: Path:
['192.168.123.3', '192.168.123.7', '192.168.124.8']
_install path: mac_src: fc:f8:ae:5e:dc:e7 mac_dst: 33:33:ff:dd:ee:ff
Installing in switch 192.168.123.3 in port 1 output port 3 src MAC dst
    fc:f8:ae:5e:dc:e7 dst MAC addr 33:33:ff:dd:ee:ff
--- PushPORT
Send Push Action with output 3
Installing in switch 192.168.123.7 in port 4 output port 1 src MAC dst
    fc:f8:ae:5e:dc:e7 dst MAC addr 33:33:ff:dd:ee:ff
--- PushPORT
Send Push Action with output 1
Switch 192.168.124.8 not openflow controlled
============> install_path: 0.00274610519409 seconds
_install path: mac_src: fc:f8:ae:5e:dc:e7 mac_dst: aa:bb:cc:dd:ee:ff
Installing in switch 192.168.123.3 in port 1 output port 3 src MAC dst
    fc:f8:ae:5e:dc:e7 dst MAC addr aa:bb:cc:dd:ee:ff
--- PushPORT
Send Push Action with output 3
Installing in switch 192.168.123.7 in port 4 output port 1 src MAC dst
    fc:f8:ae:5e:dc:e7 dst MAC addr aa:bb:cc:dd:ee:ff
--- PushPORT
Send Push Action with output 1
Switch 192.168.124.8 not openflow controlled
============> install_path: 0.00245881080627 seconds
_install path: mac_src: fc:f8:ae:5e:dc:e7 mac_dst: 33:33:00:00:00:02
Installing in switch 192.168.123.3 in port 1 output port 3 src MAC dst
    fc:f8:ae:5e:dc:e7 dst MAC addr 33:33:00:00:00:02
--- PushPORT
Send Push Action with output 3
Installing in switch 192.168.123.7 in port 4 output port 1 src MAC dst
    fc:f8:ae:5e:dc:e7 dst MAC addr 33:33:00:00:00:02
--- PushPORT
Send Push Action with output 1
Switch 192.168.124.8 not openflow controlled
============> install_path: 0.00246405601501 seconds
IPv6: ICMP6 - Path installed!
============> _packet_in_handler: 0.0180339813232 seconds
```

For each direction of the communication (MN to DMM GW and viceversa), the CLC installs three different paths:

- src MAC dst fc:f8:ae:5e:dc:e7 dst MAC addr 33:33:ff:dd:ee:ff

- src MAC dst fc:f8:ae:5e:dc:e7 dst MAC addr aa:bb:cc:dd:ee:ff
- src MAC dst fc:f8:ae:5e:dc:e7 dst MAC addr 33:33:00:00:00:02

The first path corresponds to the directed multicast layer 2 address used by IPv6 Neighbor Discovery. The second address corresponds to unicast communication between the MN (fc:f8:ae:5e:dc:e7) and the DMM GW (aa:bb:cc:dd:ee:ff). Finally the third rule corresponds to the destination multicast address used for Neighbor Solicitation messages. Through the setup of these three paths, the MN is able to contact the DMM GW and transport packets up to it. After this step, no more packets originated at the MN are received by the CLC while the MN is attached to the same access point.

One detail of the implementation worth highlighting is the fact that the matching configured in the data path elements is based on MAC address at the moment. The design of matching based on MAC addresses has an important caveat, it does not allow the use of different paths for traffic with the same origin and destination, since all flows will have the same source and destination addresses. The reason for choosing this implementation option is that our hardware currently only supports OpenFlow 1.0, and this version does not support IPv6 matches. We aim at updating the code to OpenFlow 1.4 as soon as the hardware supports it.

4.2.4 Intra-district handover

In this section we briefly explain the different steps performed by the CLC to control the movement of an MN within the same district. The flow diagram, highlighting the difference in behavior, is presented in Figure 4.6 while the output of the CLC while managing an intra-district handover is shown in Listing 4.5.

Listing 4.5: Intra-district Handover

```
EVENT ofp_event->SimpleSwitch EventOFPPacketIn

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Message received from AP switch with address: 192.168.123.4 in port 1
--------------------------------------------------------------------
mac_src: fc:f8:ae:5e:dc:e7, mac_dst: ff:ff:ff:ff:ff:ff
--------------------------------------------------------------------
ACL for Mobile Nodes is disabled.
Packet type is LLC_SNAP
Destination Mac: ff:ff:ff:ff:ff:ff
MN is already in the BCE
HANDOVER INTRA-DISTRICT
Packet content
{'sent_time': 0, 'fields': {'dsap': 0, 'ssap': 1, 'ctrl': 175}, 'aliastypes':
    [<class 'scapy.layers.l2.LLC'>], 'post_transforms': [], 'underlayer': <Dot3
    dst=ff:ff:ff:ff:ff:ff src=fc:f8:ae:5e:dc:e7 len=6 |<LLC dsap=0x0 ssap=0x1
    ctrl=175 |<Raw load='\x81\x01\x00' |>>>, 'fieldtype': {'dsap': <Field
    (LLC).dsap>, 'ssap': <Field (LLC).ssap>, 'ctrl': <Field (LLC).ctrl>}, 'time':
    1401880573.202109, 'initialized': 1, 'overloaded_fields': {}, 'packetfields':
    [], 'payload': <Raw load='\x81\x01\x00' |>, 'default_fields': {'dsap': 0,
    'ssap': 0, 'ctrl': 0}}
NEW AP:192.168.123.4 corrected in the BCE for MN:fc:f8:ae:5e:dc:e7
    -->CRC knows MN, it's in this district
Controller advertised
IPv6: ICMP6 - Router advertisement prefix: 2101::
```

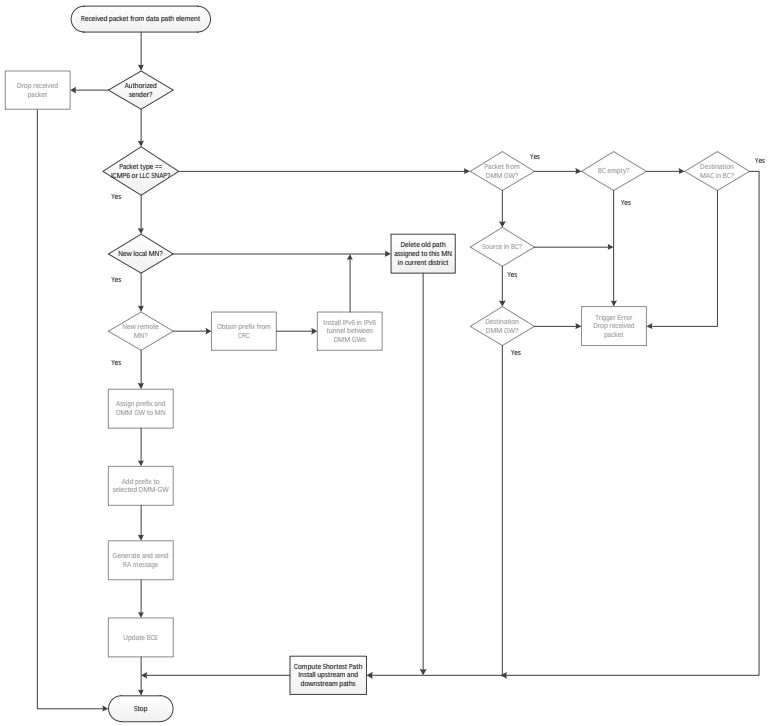


Figure 4.6: Intra-district handover, CLC

```
IPv6: ICMP6 - Sent RA
IPv6: Compute NEW route from 192.168.123.4 to 192.168.124.8
['192.168.123.4', '192.168.123.7', '192.168.124.8']
IPv6: Checking old entries in switches not in current datapath
IPv6: Compute route from 192.168.123.3 to 192.168.124.8
IPv6: OLD PATH:
['192.168.123.3', '192.168.123.7', '192.168.124.8']
Removing old entries from 192.168.123.3
Removing old entries from 192.168.123.7
Switch 192.168.124.8 not openflow controlled
Removing old entries from 192.168.123.3
Removing old entries from 192.168.123.7
Switch 192.168.124.8 not openflow controlled
Removing old entries from 192.168.123.3
Removing old entries from 192.168.123.7
Switch 192.168.124.8 not openflow controlled
_install path: mac_src: fc:f8:ae:5e:dc:e7 mac_dst: 33:33:ff:dd:ee:ff
Installing in switch 192.168.123.4 in port 1 output port 2 src MAC dst
    fc:f8:ae:5e:dc:e7 dst MAC addr 33:33:ff:dd:ee:ff
--- PushPORT
Send Push Action with output 2
Installing in switch 192.168.123.7 in port 2 output port 1 src MAC dst
    fc:f8:ae:5e:dc:e7 dst MAC addr 33:33:ff:dd:ee:ff
--- PushPORT
Send Push Action with output 1
Switch 192.168.124.8 not openflow controlled
============> install_path: 0.00144004821777 seconds
_install path: mac_src: fc:f8:ae:5e:dc:e7 mac_dst: aa:bb:cc:dd:ee:ff
```

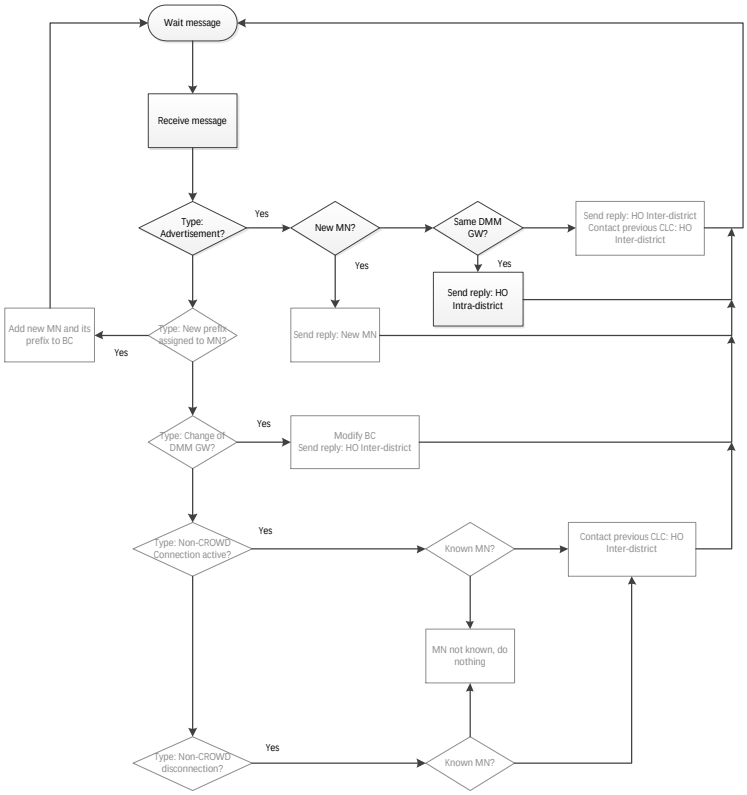


Figure 4.7: Intra-district handover, CRC

```
Installing in switch 192.168.123.4 in port 1 output port 2 src MAC dst
    fc:f8:ae:5e:dc:e7 dst MAC addr aa:bb:cc:dd:ee:ff
--- PushPORT
Send Push Action with output 2
Installing in switch 192.168.123.7 in port 2 output port 1 src MAC dst
    fc:f8:ae:5e:dc:e7 dst MAC addr aa:bb:cc:dd:ee:ff
--- PushPORT
Send Push Action with output 1
Switch 192.168.124.8 not openflow controlled
============> install_path: 0.00135898590088 seconds
IPv6: Updating BCE from fc:f8:ae:5e:dc:e7, IP: 192.168.123.4 to 192.168.123.4
IPv6: Result update 192.168.123.4
IPv6: ICMP6 - Path installed!
---------------------------------------------------------------------

DEBUG INFO:
---------------------------------------------------------------------
Received packet is LLC_SNAP
Old MN for this district (HANDOVER INTRADISTRICT).
Updated upward path :
 From MN (fc:f8:ae:5e:dc:e7) to GW(33:33:ff:dd:ee:ff and aa:bb:cc:dd:ee:ff)
Updated AP(192.168.123.4) associated with the MN mac(fc:f8:ae:5e:dc:e7) in BCE
---------------------------------------------------------------------
============> _packet_in_handler: 0.0195059776306 seconds
```

The intra-district handover starts by the MN associating to a different PoA of the same district. This fact is acknowledged by the CLC through the reception of an LLC message, following the same behavior as in the initial attachment of the MN to the district. At this point of time, the CLC checks internally if the MN is recognized (has been attached to this district). In this case, this check is successful and the CLC retrieves the information regarding the previous PoA the MN was attached to. Since this PoA is different from the one registered in the BC, the CLC assumes the handover corresponds to the intra-district case and informs the CRC. The CRC flow diagram can be seen in Figure 4.7, while its output console is shown in Listing 4.6. To speed up the handover process, the CLC sends a Router Advertisement to the MN as soon as it verifies the handover type. This reduces the handover delay since the MN does not need to send a Router Solicitation message. Finally, the CLC removes the old paths and installs a new set of paths joining the new AP to the DMM GW.

Listing 4.6: CRC detection of intra-district handover

```
Incoming packet:
  adv_type = A
  mn_mac = fc:f8:ae:5e:dc:e7
  prefix = 00
  prefixlen = 0
  default_gw = 2001:720:410:1001:20d:b9ff:fe2e:9855
  controller = 192.168.123.1

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Message received in port 6633
------------------------------------------------------------------
MAC: fc:f8:ae:5e:dc:e7
Advertisement: Do you know fc:f8:ae:5e:dc:e7 ?
Answer: Old MN
HANDOVER INTRANETWORK

CACHE:
--------------------------------------------------------------------------------
MAC: fc:f8:ae:5e:dc:e7 Prefix: 2101::/64 DW: 2001:720:410:1001:20d:b9ff:fe2e:9855
--------------------------------------------------------------------------------
```

4.2.5 Inter-district Handover

This section explains how the DMM implementation works when there is an Inter-district handover. This is the most complex case since it involves different logical entities at the different controllers communicating asynchronously between each other. As explained in Section 4.1 the design of the Ryu framework is based on events and as such the implementation of servers or just functions polling resources is complex. Once an MN hands off to a different district, the previous CLC has no mechanisms of knowing that the MN has left, hence all previous status and forwarding rules remain as if the MN was still attached to the district. In order to inform the previous CLC of the new location of the MN, we need a permanent connection between the CRC and CLC. This permanent connection has been implemented within the Ryu framework as a separate listening thread, which is in charge of receiving the commands from the CRC and interface with the DMM-GW in order to install and configure an IPv6-in-IPv6

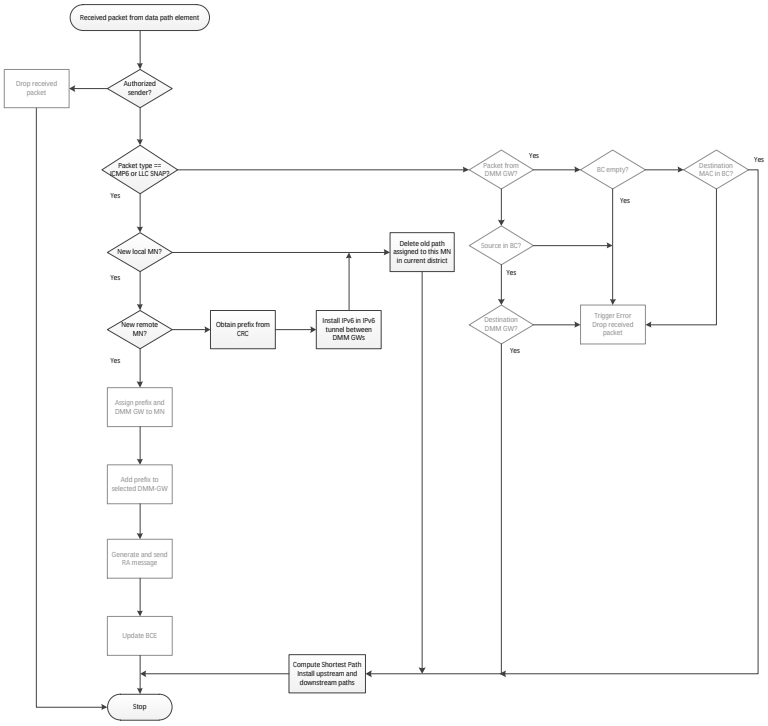


Figure 4.8: Inter-district handover, new CLC

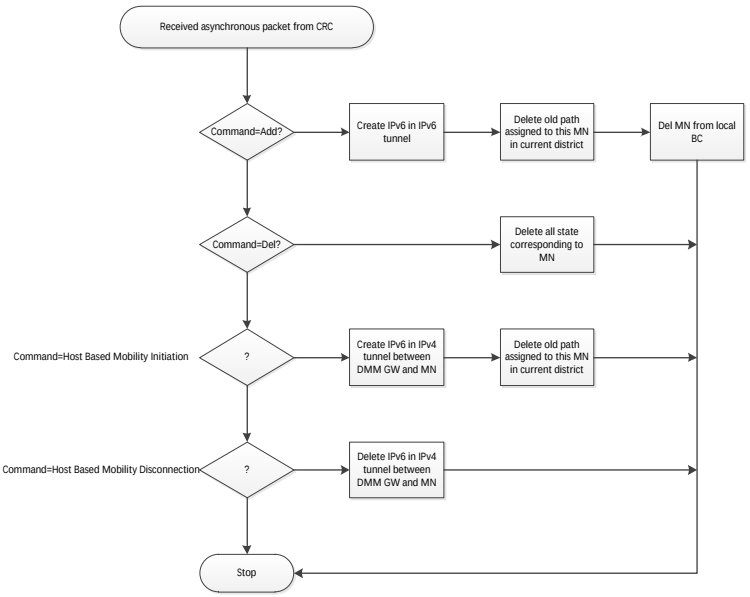


Figure 4.9: Inter-district handover, old CLC, listening thread

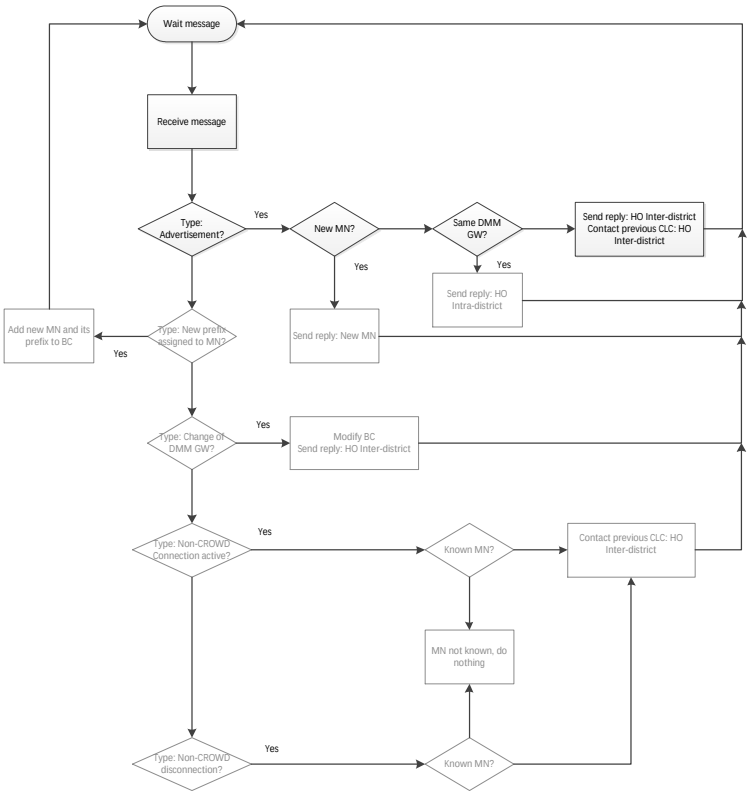


Figure 4.10: Inter-district handover, CRC

tunnel with the new DMM-GW. Figure 4.8 shows the flow diagram of the new CLC that is receiving the MN in the new district. Figure 4.9 presents the flow diagram of the separated thread implementing the continuous connection with the CRC and finally Figure 4.10 depict the behavior of the CRC.

Listing 4.7: Inter-district handover (CLC)

```
Message received from AP switch with address: 192.168.123.14 in port 1
----------------------------------------------------------------------
mac_src: fc:f8:ae:5e:dc:e7, mac_dst: ff:ff:ff:ff:ff:ff
----------------------------------------------------------------------
ACL for Mobile Nodes is disabled.
Packet type is LLC_SNAP
Destination Mac: ff:ff:ff:ff:ff:ff
Mobile Node IS NOT in the BCE
Packet content
{'sent_time': 0, 'fields': {'dsap': 0, 'ssap': 1, 'ctrl': 175}, 'aliastypes':
    [<class 'scapy.layers.l2.LLC'>], 'post_transforms': [], 'underlayer': <Dot3
    dst=ff:ff:ff:ff:ff:ff src=fc:f8:ae:5e:dc:e7 len=6 |<LLC dsap=0x0 ssap=0x1
    ctrl=175 |<Raw load='\x81\x01\x00' |>>>, 'fieldtype': {'dsap': <Field
    (LLC).dsap>, 'ssap': <Field (LLC).ssap>, 'ctrl': <Field (LLC).ctrl>}, 'time':
    1401880557.262594, 'initialized': 1, 'overloaded_fields': {}, 'packetfields':
    [], 'payload': <Raw load='\x81\x01\x00' |>, 'default_fields': {'dsap': 0,
    'ssap': 0, 'ctrl': 0}}
EVENT ofp_event->SimpleSwitch EventOFPPacketIn
    -->CRC knows MN, it's in another district
HANDOVER INTER-DISTRICT (NEW DISTRICT)
```

```
INSTALL TUNNEL
ip -6 ne flush dev eth0; ip -6 tunnel add Tun0 mode ip6ip6 local
    2001:720:410:1001:20d:b9ff:fe2e:9805 remote
    2001:720:410:1001:20d:b9ff:fe2e:9855 dev eth1; ip link set dev Tun0 up;
Deleting DESCENDING path from GW:192.168.124.18 to AP:192.168.123.14para el
    NM:fc:f8:ae:5e:dc:e7
Removing old entries from 192.168.123.14
Removing old entries from 192.168.123.6
Switch 192.168.124.18 not openflow controlled
Deleting downward path finished
Type: Y Prefix: 2101:: mask: 64 defGW: 2001:720:410:1001:20d:b9ff:fe2e:9855
IPv6: ICMP6 - MAC fc:f8:ae:5e:dc:e7 added to BCE.
IPv6: ICMP6 - Router advertisement prefix: 2101::
ip -6 route add 2101::/64 dev eth0
IPv6: ICMP6 - Sent RA
============> _packet_in_handler: 0.0153210163116 seconds
```

We start with the detailed description of this scenario with Listing 4.7. The output of the new CLC is very similar to the one shown when a new MN is attached to the network until the handover case is recognized by the CRC. In this case, first the new CLC receives the prefix information together with the address of the previous DMM GW from the CRC (see Listing 4.8). Before sending any RA to the MN or setting data paths, the CLC talks with the DMM GW and setups an IPv6-in-IPV6 tunnels between the new and old DMM GWs. It is important to highlight that the new CLC can only configure one end-point of the tunnel (the new DMM GW), but the end-point corresponding to the previous DMM GW must be configured by the previous CLC.

Currently we are working towards the implementation of a better DMM GW - CLC interface, but in the test performed at this moment we are using a remote command application based on SSH. Basically the configuration performed in the new DMM GW can be summarized as follows:

```
....
ip -6 ne flush dev eth0; ip -6 tunnel add Tun0 mode ip6ip6 local
    2001:720:410:1001:20d:b9ff:fe2e:9805 remote 2001:720:410:1001:20d:b9ff:fe2e:9855 dev
    eth1; ip link set dev Tun0 up;
....
```

After creating the local end point of the tunnel, the CLC sends an RA to the MN, containing the same prefix information as used in the previous district. After that, the CLC proceeds to the creation of down/upstream paths, which are not shown here to reduce the size of this document.

Listing 4.8: Inter-district handover (CRC)

```
Incoming packet:
  adv_type = A
  mn_mac = fc:f8:ae:5e:dc:e7
  prefix = 00
  prefixlen = 0
  default_gw = 2001:720:410:1001:20d:b9ff:fe2e:9805
  controller = 192.168.123.5

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Message received in port 6633
----------------------------------------------------------------
MAC: fc:f8:ae:5e:dc:e7
```

```
Advertisement: Do you know fc:f8:ae:5e:dc:e7 ?
Answer: Old MN
HANDOVER INTERNETWORK
DONE

CACHE:
------------------------------------------------------------------------------
MAC: fc:f8:ae:5e:dc:e7 Prefix: 2101::/64 DW: 2001:720:410:1001:20d:b9ff:fe2e:9805
------------------------------------------------------------------------------
```

In parallel to the above operations, the CRC contacts the previous CLC and informs it of the handover event and the new location of the MN. The output of the CLC listener thread can be found in Listing 4.9. Upon reception of a message from the CRC indicating the need to configure an inter-district handover, the listener thread contacts the DMM GW in order to configure the local end-point of the tunnel. In addition, this thread is in charge of setting the correct set of routes to forward the MNs' traffic through the tunnel. Finally, the CLC listener thread will interact with the data path elements and remove all the paths configured for this MN in the local district, since they are not longer used.

Listing 4.9: Inter-district handover (CLC listener thread)

```
Received packet in the rectifier
ip -6 tunnel add Tun0 mode ip6ip6 local 2001:720:410:1001:20d:b9ff:fe2e:9855 remote
    2001:720:410:1001:20d:b9ff:fe2e:9805 dev eth1; ip link set dev Tun0 up; ip -6
    rule add to 2101::/64 table myorg; ip -6 route add default dev Tun0 table myorg
Deleting UPWARD path from AP:192.168.123.3 to GW:192.168.124.8
Removing old entries from 192.168.123.3
Removing old entries from 192.168.123.7
Switch 192.168.124.8 not openflow controlled
Deleting UPWARD path finished
Deleted MN fc:f8:ae:5e:dc:e7 from BCE
------------------------------------------------------------
Received packet in the rectifier:
 Order: A
 Name: Tun0
 Origin: 2001:720:410:1001:20d:b9ff:fe2e:9855
 Destiny: 2001:720:410:1001:20d:b9ff:fe2e:9805
 Prefix: 2101::
 Lenght: 64
 NM: fc:f8:ae:5e:dc:e7
 Direction: T
Creating tunnel Tun0 on 192.168.123.77
Deleted UPWARD path from MN to GW
------------------------------------------------------------
```

4.2.6 Host-based Mobility

Host-based part of the Connectivity Management solution consists in the improvement of the network-based part. Besides the functionality described above in the beginning of the present chapter (Chapter 4) the implementation of the host-based solution brings the possibility of attaching the mobile node to a completely independent IPv4 network. To achieve this functionality we have developed an application to manage

the mobile node interaction. This software consists on a graphic user interface which allows user to connect with every access points of the testbed.

To test this host-based approach, we have to do the following steps. First of all we attach the mobile node to one of our AP controlled by Open vSwitch. Once all the steps have been performed in the network as are displayed in sub section 4.2.3, the user can connect with a totally independent IPv4 network. To do this, user selects the IPv4 network from the scan section of the Mobile Node application (Fig. 7.3) and the script will do the needed logic operations to keep connectivity. These operations are the following:

1. Connect the wireless interface with the Access Point of the new network
2. Launch dhclient script to get a IPv4 direction from the new network
3. Send the IPv4 direction assigned and the Mac Address of the Mobile Node to the CROWD Regional Controller
4. Receive from the CROWD Regional Controller the IPv4 direction of the Gateway which is linked with the mobile node
5. Create an IPv6 in IPv4 tunnel between IPv4 of the Mobile node and IPv4 of the Default Gateway
6. Route all traffic generate through the tunnel created

Besides the steps followed by the Mobile Node there are some operations which must be done. When the CROWD Regional Controller receive the IPv4 of the Mobile Node, it sends a message to the CROWD Local Controller of the old district in which the Mobile Node was anchored. When it happens, the CROWD Local Controller sends a message to the old Default Gateway of the Mobile Node to create the other end of the IPv6 in IPv4 tunnel.

```
....
ip tunnel add tun6in4 mode sit remote 163.117.139.80 local 163.117.140.71 ttl 255; ip link
    set tun6in4 up; ip -6 rule add to 2101::49d2:8d72:54a2:6c40 table mytun; ip -6 rule add
    to 2101::/64 table mytun; ip -6 ro add default dev tun6in4 table mytun
....
```

Figure 4.11 represents all the messages sent between Mobile Node, CROWD Local Controller District 1, CROWD Regional Controller, CROWD Local Controller District 2, Default Gateway of District 1 and the IPv4 Network during the processes explained above:

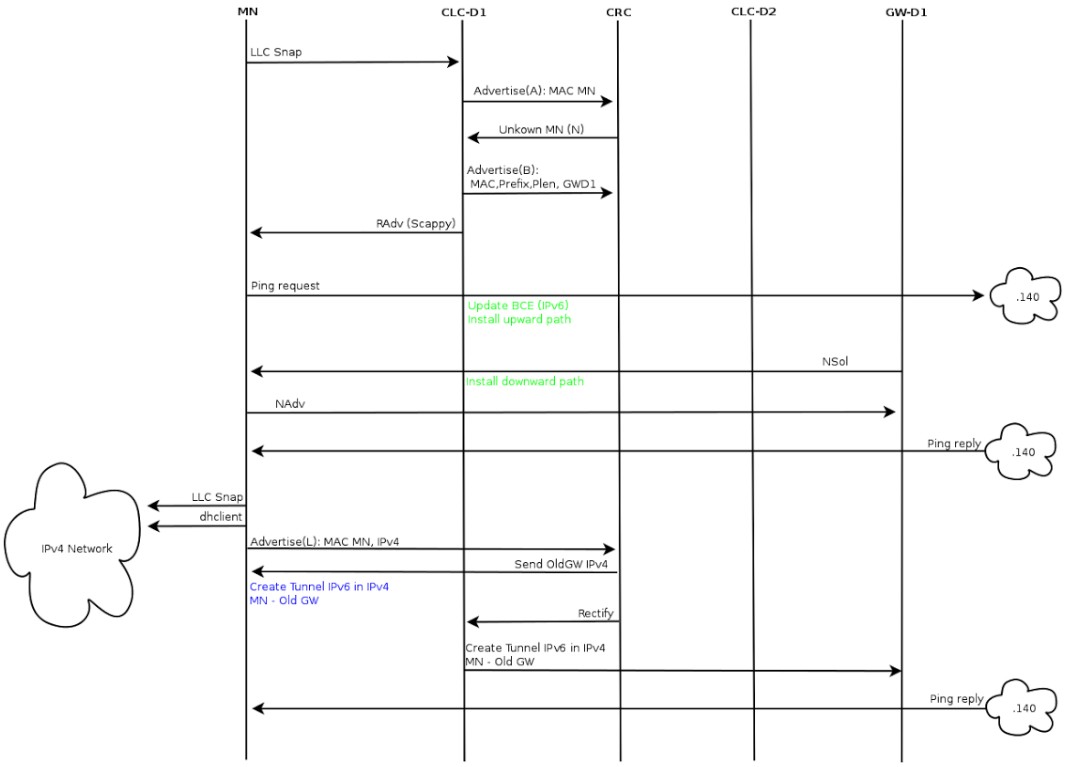


Figure 4.11: Host-based messages

Figure 4.12 includes the messages produced when Mobile Node comes back to the same old district after performing a handover with a different IPv4 network.

The independent network handover starts by the MN associating to an Access Point of another network. Then, the mobile node asks for an IPv4 direction to the dhcp server of the new network. When the mobile node obtains an IPv4 informs the CRC. The CRC output console is shown in Listing 4.10.

Listing 4.10: CRC ipv4 independent network handover

```
Incoming packet:
  adv_type = L
  mn_mac = fc:f8:ae:5e:dc:e7
  prefix = 163.117.139.80
  prefixlen = 1
  default_gw = 163.117.139.80

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Message received in port 6633
----------------------------------------------------------------------
MAC: fc:f8:ae:5e:dc:e7
LTE CONNECTION DETECTED
Found in District A
IPv4 MN: 163.117.139.80
CLC_IP: 192.168.123.1

CACHE:
------------------------------------------------------------------------------------
MAC: fc:f8:ae:5e:dc:e7 Prefix: 2101::/64 DW: 2001:720:410:1001:20d:b9ff:fe2e:9855
------------------------------------------------------------------------------------
```

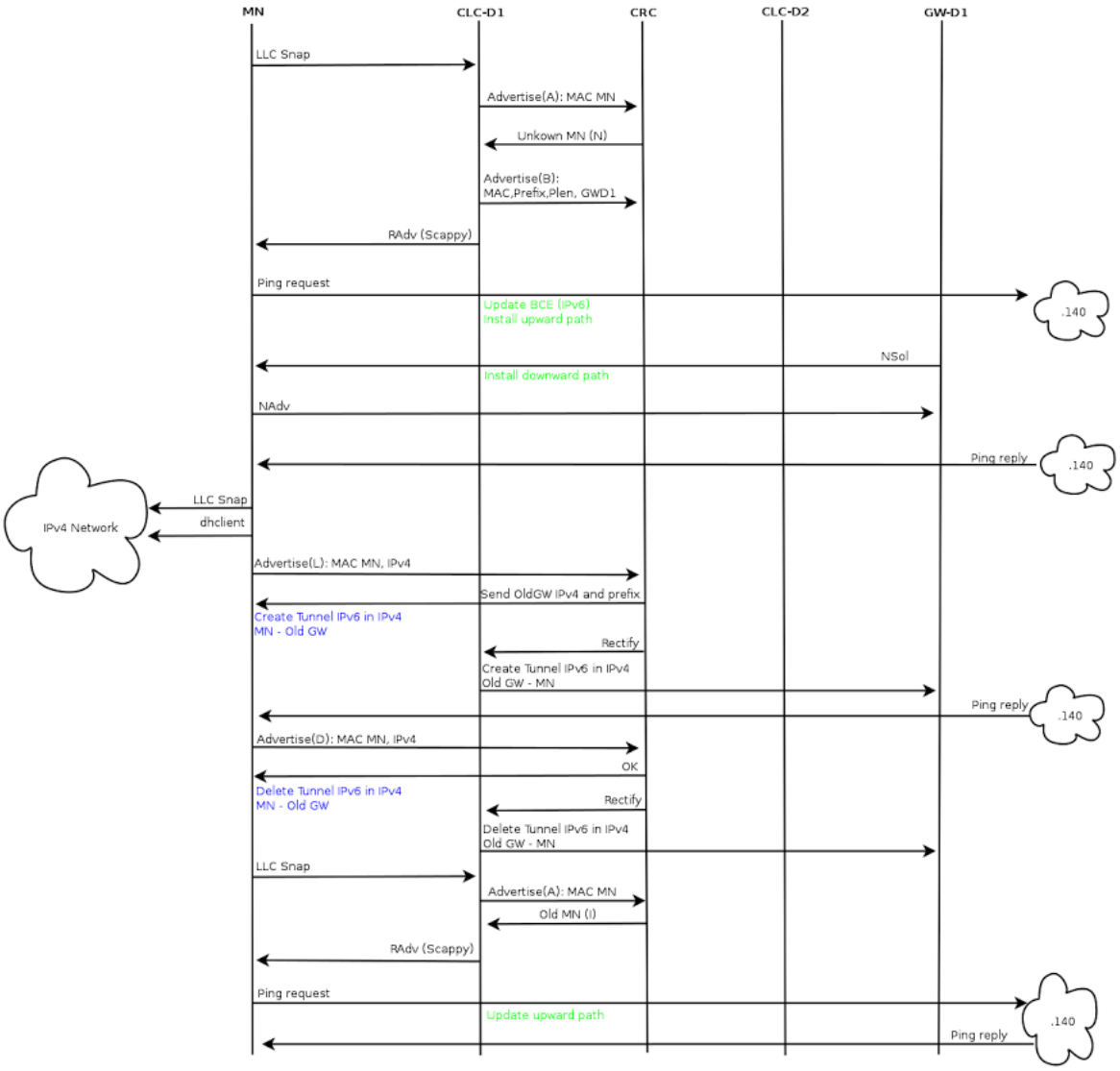


Figure 4.12: Host-based regression messages

When the CRC contacts the previous CLC and informs it of the handover event with an independent IPv4 network. The output of the CLC listener thread can be found in Listing 4.11. Upon reception of a message from the CRC indicating the need to configure an IPv6 in IPv4 tunnel, the listener thread contacts the DMM GW in order to configure the local end-point of the tunnel. In addition, this thread is in charge of setting the correct set of routes to forward the MNs' traffic through the tunnel.

Listing 4.11: Inter-district handover (CLC listener thread)

```
Received packet in the rectifier
IPv4 Independent Network Connection Detected
None of the installed paths were deleted
ip tunnel add tun6in4 mode sit remote 163.117.139.80 local 163.117.140.71 ttl 255;
    ip link set tun6in4 up; ip -6 rule add to 2101::49d2:8d72:54a2:6c40 table
    mytun; ip -6 rule add to 2101::/64 table mytun; ip -6 ro add default dev
    tun6in4 table mytun
------------------------------------------------------
Received packet in the rectifier:
```

```
 Order: L
 Name: Lte0
 Origin: 163.117.140.71
 Destiny: 163.117.139.80
 Prefix: 2101::
 Lenght: 64
 NM: fc:f8:ae:5e:dc:e7
 Direction: T
LTE Connection Detected
Creating tunnel Lte0 on 192.168.123.77
-------------------------------------------------------
```

CHAPTER 5

Experiments

This chapter will explain the experiments performed during the realization of this final degree project. In this chapter there exist some CDF Graphics. The cumulative distribution function (CDF), [19] describes the probability that a real-valued random variable X with a given probability distribution will be found to have a value less than or equal to x. In the case of a continuous distribution, it gives the area under the probability density function from minus infinity to x. With the help of these graphics we can analyze the time results of our experiments. To ensure the correct analysis, we have repeated all these measurements 30 times resetting the whole testbed 7.4.

5.1 Handover processes

To measure the timing taken during an intra-district handover we have send a flow of echo packets through the OpenFlow network (from a districts' AP to its DMM Gateway), and then we have performed an intra-district handover and counted the packets lost. With that number and the timing between packets we can estimate how much time takes the system to recover connection during a handover. The process for the inter-district handover consists in attach the mobile node to the district A, then start generating traffic to its DMM Gateway, continue doing the handover to the district B and finally creating a tunnel between both DMM Gateways. The statistical results are the following (Fig.5.1):

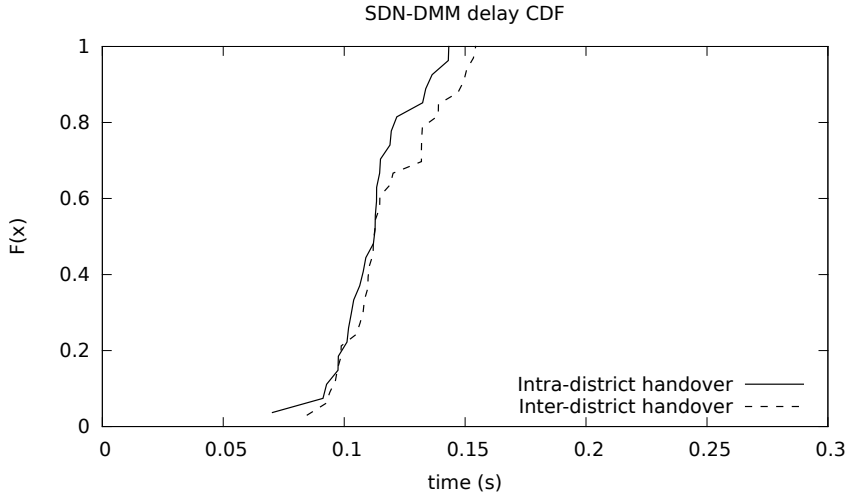


Figure 5.1: SDN-DMM delay CDF - Intra- and Inter-district Handover

Figure 5.2 represents the host-based handover process through an independent IPv4 network. As we can see, there is a significant time difference between this CDF and Intra- and Inter- handover CDF 5.1. We can explain this because in the independent IPv4 network handover we add some new operations as the creation of IPv6 in IPv4 tunnel.

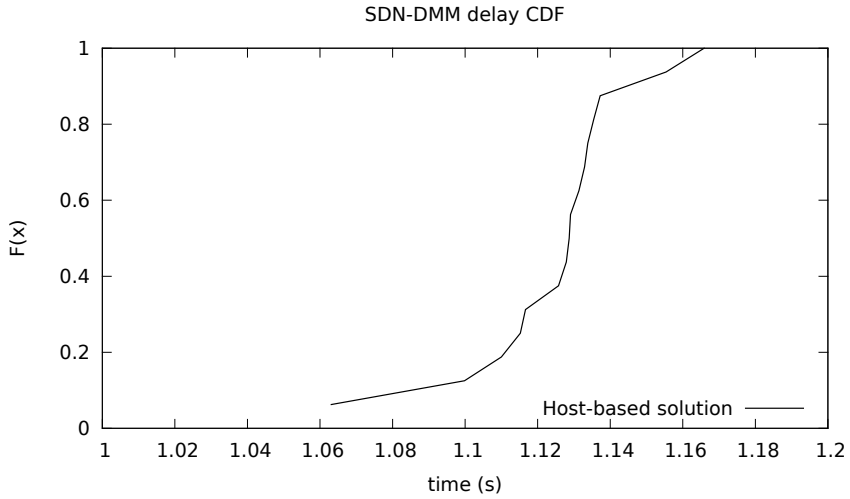


Figure 5.2: SDN-DMM delay CDF - IPv4 Network

5.2 Effect of the CLC-CRC delay

These experiments simulate the proximity or remoteness of some of the controllers of the experiment. To simulate these experiments we have use the Linux Advanced Routing And Traffic Control application. An example of adding a delay of 25 milliseconds:

```
1    $ sudo /sbin/tc qdisc add dev eth1 root netem delay 25ms
```

5.2.1 Effect of increasing the delay between the data-path elements and the CRC

First we add some delay to the CROWD Regional Controller trying to simulate distance between itself and each CROWD Local Controller.

Intra-district Handover

Figure 5.3 represents the intra-network handover process with a delay added to the CROWD Regional Controller from 25 ms to 500 ms. As we can see, there is a direct relation between the delay of the SDN and the delay added to the CRC.

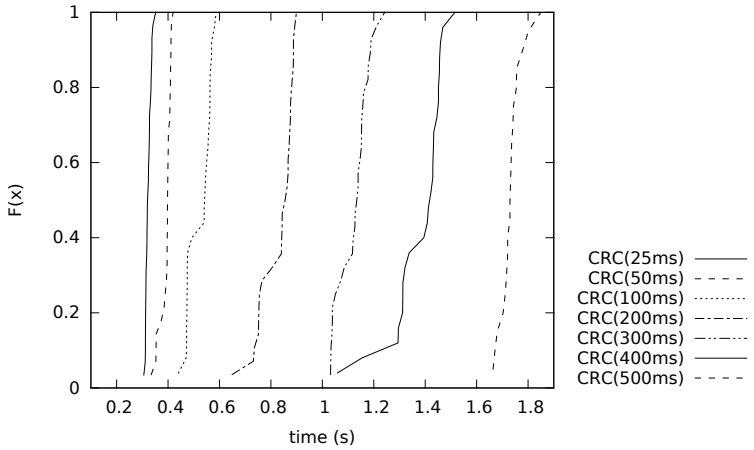


Figure 5.3: Intra-district Handover.

Inter-district Handover

Figure 5.3 represents the handover inter-network process with a delay added to the CROWD Regional Controller from 25 ms to 500 ms. As we can see, there is a direct relation between the delay of the SDN and the delay added to the CRC.

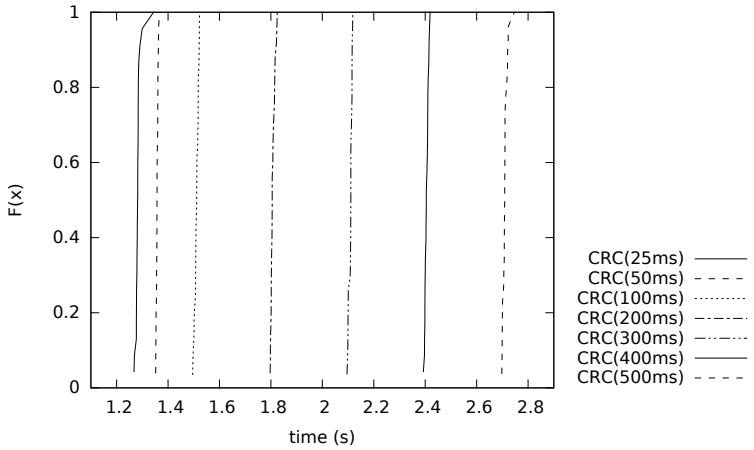


Figure 5.4: Inter-district Handover.

5.2.2 Effect of increasing the delay between the data-path elements and the CLC

Now we add some delay to the each CROWD Local Controller trying to simulate distance between itself and the other CROWD Local Controller, with the CROWD Regional Controller and between all data-path elements and CROWD Local Controllers.

Intra-district Handover

Figure 5.5 represents the handover inter-network process with a delay added to the CROWD Regional Controller from 25 ms to 500 ms. As we can see, there is a direct relation between the delay of the SDN and the delay added to the CRC. This relation is stronger than the relation observed when adding delay to the CROWD Regional Controller seen in section 5.2.1 and this is due of the amount of packets transmitted from each CROWD Local Controller during the handover process.

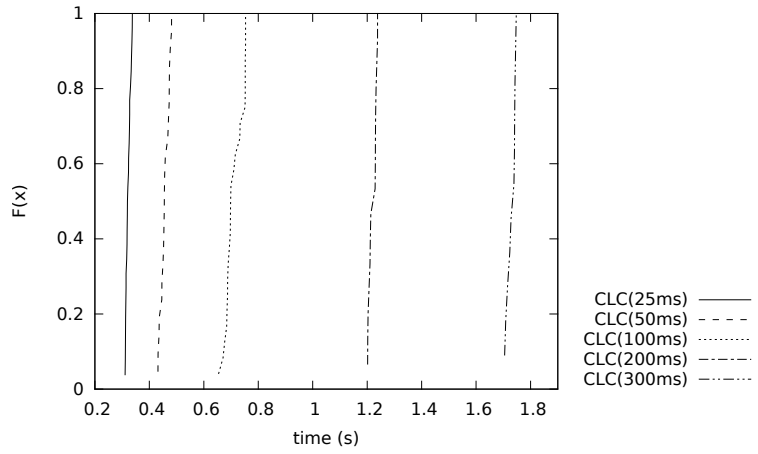


Figure 5.5: Intra-district Handover.

Inter-district Handover

Figure 5.6 represents the handover inter-network process with a delay added to the CROWD Local Controller from 25 ms to 500 ms.

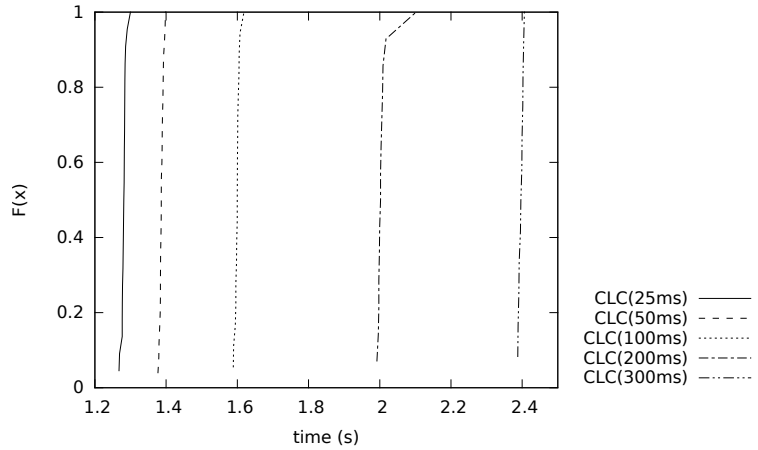


Figure 5.6: Inter-district Handover.

5.2.3 Effect of the combinations of delays CRC/CLC and data-path elements/CLC

This last experiment adds delay to every Controller of the testbed.

Intra-district Handover

Figure 5.7 represents the handover intra-network process with a delay added to the CROWD Local Controller and to the CROWD Local Controller from 25 ms to 300 ms

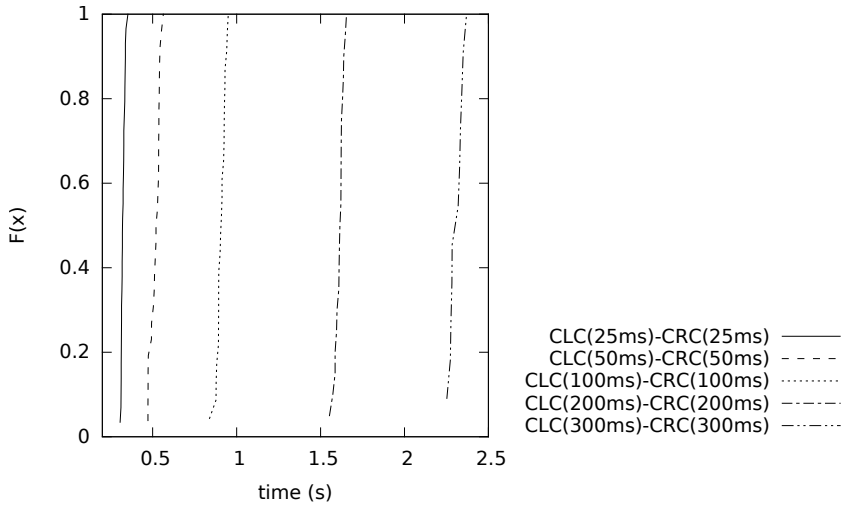


Figure 5.7: Intra-district Handover.

Inter-district Handover

Figure 5.8 represents the handover intra-network process with a delay added to the CROWD Local Controller and to the CROWD Local Controller from 25 ms to 300 ms

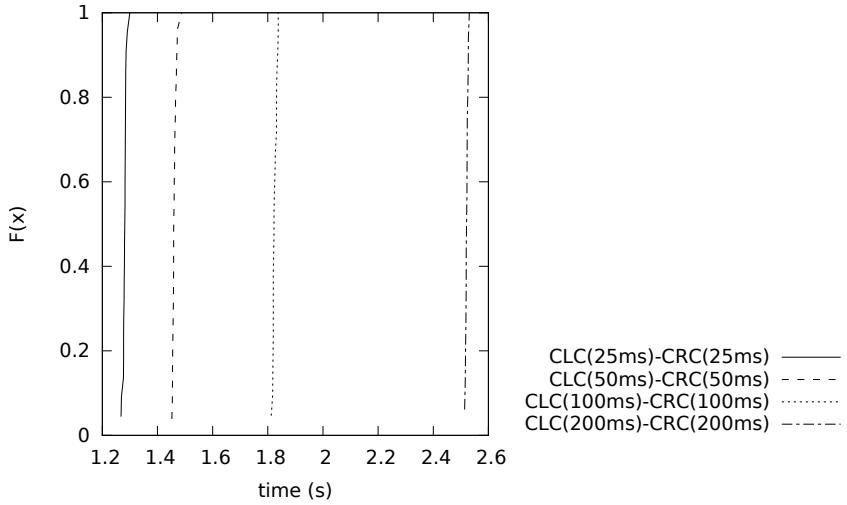


Figure 5.8: Inter-district Handover.

5.3 Characterization

In this section we are going to characterize the times spent during the different performance of the CROWD Local Controller.

First of all, in table 5.1 we are going to analyze the time spent during the treatment of the LLC_SNAP packet generated when the mobile node connect with an access point of the testbed. Basically, the CROWD Local Controller do the following operations: perform signaling with CRC, store MN in the BCE, send Router advertisement [11] and perform signaling with CRC again.

LLC_SNAP	
0,0250840187	
0,0193049908	
0,0229609013	
0,0256490707	
0,019657135	
0,0216917992	
0,0191061497	
0,0243210793	
0,0190489292	
0,0197660923	
Average:	0,02165901661
Median:	0,0207289457
Min:	0,0190489292
Max:	0,0256490707
Percentile 25:	0,0193930268
Percentile 75:	0,0239810348

Table 5.1: Process LLC_SNAP packet

Table 5.2 represents the time spent during the treatment of the IPv6 packet sent by the Mobile Node to the DMM Gateway when mobile node sends an echo request to its own DMM Gateway. Basically, the CROWD Local Controller do the following operations: perform signaling with CRC, install upward path and update BCE.

IPv6 (MN to GW)	
0,0208730698	
0,0086619854	
0,0087020397	
0,0168080330	
0,0086381435	
0,0147469044	
0,0098221302	
0,0175299644	
0,0086820126	
0,0086791515	
Average:	0,0123143434
Median:	0,0092620850
Min:	0,0086381435
Max:	0,0208730698
Percentile 25:	0,0086798668
Percentile 75:	0,0162927508

Table 5.2: Process IPv6 packet from MN to GW (Ping request)

Table 5.3 represents the time spent during the treatment of the IPv6 packet sent by the DMM Gateway to Mobile Node when the Gateway sends a neighbor solicitation message to the mobile node. Basically, the CROWD Local Controller do the following operations: perform signaling with CRC, install upward path and update BCE.

IPv6 (GW to MN)	
0,0165150166	
0,0075590611	
0,0076041222	
0,0076379776	
0,0075581074	
0,0122618675	
0,0077269077	
0,0075709820	
0,0076000690	
0,0075869560	
Average:	0,0089621067
Median:	0,0076020956
Min:	0,0075581074
Max:	0,0165150166
Percentile 25:	0,0075749755
Percentile 75:	0,0077046752

Table 5.3: Process IPv6 packet from GW to MN (Neighbor Solicitation)

Table 5.4 represents the time spent during the treatment of the IPv6 packet sent by the Mobile Node to the DMM Gateway when the mobile node performs an Intra-district handover. Basically, the CROWD Local Controller do the following operations: perform signaling with CRC, update MN in the BCE, send Router advertisement [11] and update upward path.

INTRANetwork	
	0,0316460133
	0,0197629929
	0,0199420452
	0,0197811127
	0,0199420452
	0,0258381367
	0,0194652081
	0,0198559761
	0,0197818279
	0,0190780163
Average:	0,0215093374
Median:	0,0197818279
Min:	0,0190780163
Max:	0,0258381367
Percentile 25:	0,0197629929
Percentile 75:	0,0199420452

Table 5.4: Process LLC_SNAP packet after an INTRA-Network handover

Table 5.5 represents the time spent during the treatment of the IPv6 packet sent by the DMM Gateway to Mobile Node when the Gateway sends a neighbor solicitation message to the mobile node after the Intra-district handover process. Basically, the CROWD Local Controller do the following operations: perform signaling with CRC, install upward path and update BCE.

INTRA-IPv6 GW to MN	
0,0068140030	
0,0066909790	
0,0067369938	
0,0066840649	
0,0067009926	
0,0113999844	
0,0067451000	
0,0067698956	
0,0067570210	
0,0066821575	
Average:	0,0071981192
Median:	0,0067369938
Min:	0,0066821575
Max:	0,0113999844
Percentile 25:	0,0066909790
Percentile 75:	0,0067570210

Table 5.5: Process an IPv6 packet from GW to MN after an INTRA-Network handover

Table 5.6 represents the time spent during the treatment of the message sent by the CROWD Regional Controller to the CROWD Local Controller when Mobile Node performs an Inter-district handover. These actions take place in the new district. Basically, the CROWD Local Controller do the following operations: Install tunnel, Delete downward path and send Router advertisement [11].

INTER-Network	
0,0162341595	
0,0158882141	
0,0160510540	
0,0158059597	
0,0156180859	
0,0165131092	
0,0155749321	
0,0163249969	
0,0158932209	
0,0156250000	
Average:	0,0159528733
Median:	0,0158495903
Min:	0,0155749321
Max:	0,0165131092
Percentile 25:	0,0156232715
Percentile 75:	0,0161195397

Table 5.6: INTER-Network. New district

Finally, with all the above measurements, we can generate the following graphic to represent together all the actions characterized.

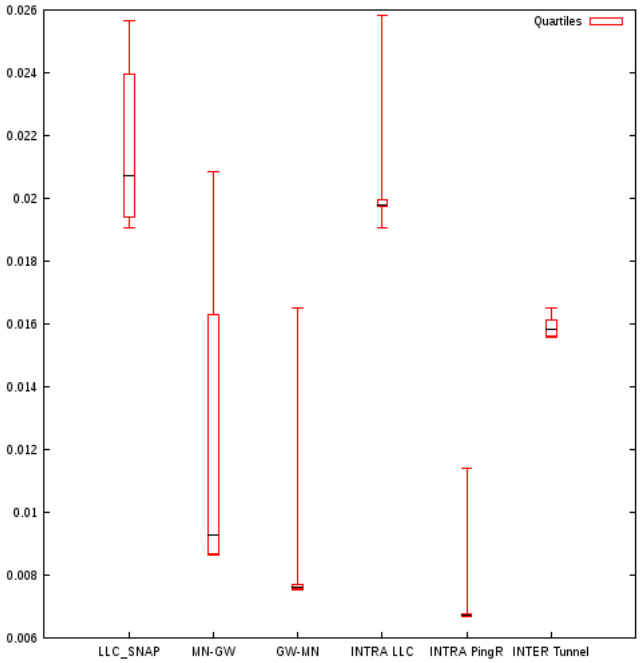


Figure 5.9: CandleStick graph of characterized measures

CHAPTER 6

Planning

This chapter explains the whole planning of this final degree project. All the work has been done during exercise of a scholarship of the Telefonica Talentum Startups Programm. These scholarships target early career professionals with a technical and technological background. The Talentum project seeks to promote employment of young people, as well as familiarize future professionals to the labor market and the real-life company environment. The scholarship lasts six months and to achieve the goals of the project three main human resources have been used: an engineer as developer, a Ph.D student as supporter and a Ph.D as the director of the project. The director is in charge of drawing the lines of action to accomplish the project. The supporter is in charge of helping the engineer to solve specific network issues thanks to her experience on the field. The engineer is in charge of the full development of the project including the software codification and the process of creating the testbed.

The development of the project was divided in five different phases:

1. **Study of the problem**: Starting from the study of the mobility issue presented in high density networks and following the possible solutions raised by the DMM study group in the CROWD project, we studied SDN as the preferred solution. After the study of different Open Flow controllers and implementations, we decided to use *Ryu* due to its great documentation and simplicity and because is coded with python. We also decided to use *Open vSwitch* as open flow implementation for the *Alix* nodes, and Pantou for the *Linksys* nodes.Finally we set up the testbed with two districts, one gateway per district, two access points per district, one CROWD Local Controller per district and one CROWD Regional Controller.

2. **Network-Based Solution**: After the assembled of the testbed we started developing the Network-Based Solution with the IPv6 implementation for the *Ryu* controller. Once finished the development, we did the network-based tests.

3. **Host-Based Solution**: This stage started only when the Network-Based Solution was tested correctly. We developed the mobile node script and modified the CROWD Controllers to achieve this functionality. When the codification was finished, we did the host-based tests.

4. **Delay Tests**: Once everything was coded, we developed a script to achieve test automation.

5. **Documentation**: This stage started with the first implementation of the Network-Based Solution and continued during the whole life of the development. The process of writing this final degree project document started when the delay tests were finished.

6.1 Time

The following picture represents the Gantt diagram of the present project:

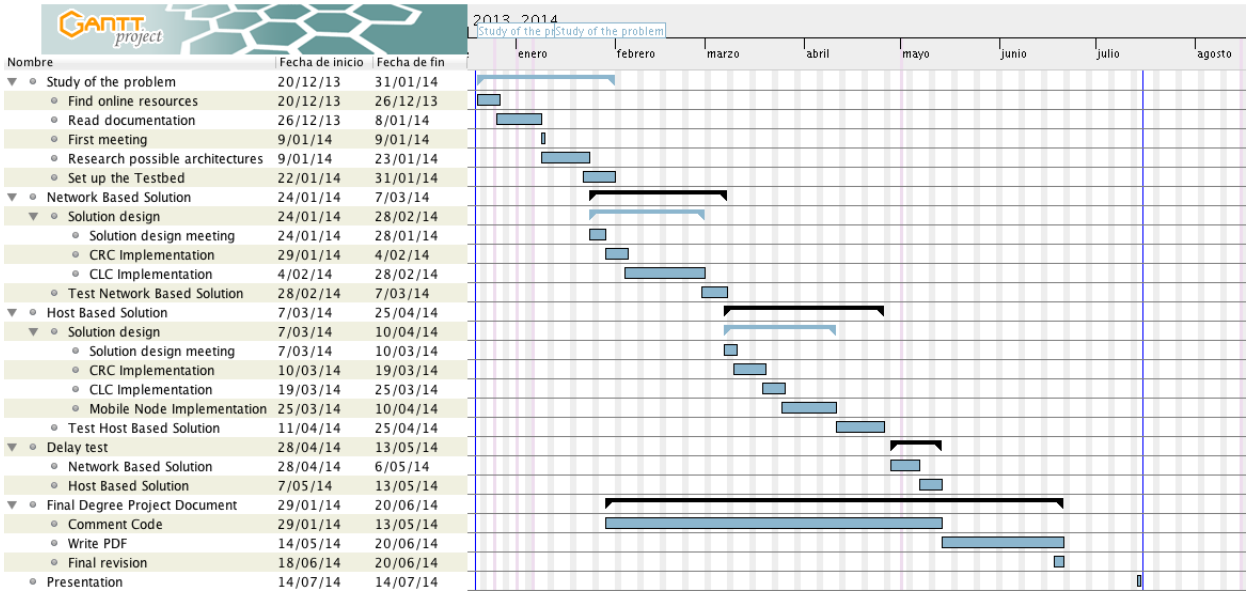


Figure 6.1: Gantt diagram

6.2 Estimation

6.2.1 Human Resources

The personnel resources estimated as necessary as accomplishment of this project were budgeted as shown in the following table:

Name	Category	Cost/Hour	Hours	Cost
Antonio de la Oliva	Senior Engineer	36	44	1584
Isabel Sanchez Bueno	Senior Engineer	36	88	3168
Pablo Castellanos Santamaría	Engineer	20	660	13200

Table 6.1: Human resources costs

6.2.2 Software costs

As all the software used to develop this project is open source (Linux Operating System, Open vSwitch, Ryu, Terminator, Vim, Dia, Textstudio), no costs has been derived for purchasing any proprietary cost license

6.2.3 Hardware costs

The next table breaks down the hardware costs incurred in the project.

Concept	**Units**	**Cost/Unit**	**Cost**
Alix 2d3	7	94	658
Linksys WRT54GL	2	50	100
Ethernet wire - Cat 5 (m)	0,55	15	8,25
D-Link 24-Port Switch	1	177	177
CLC Computer	2	300	600
CRC Computer	1	300	300
Mobile Node Laptop	1	700	700

Table 6.2: Hardware costs

6.2.4 Costs summary

This table is the summary of the total costs spent for the realization of the project.

Concept	**Total Cost**
Human resources	17952
Software	0
Hardware	2543,25

Table 6.3: Summary of total costs

As calculated above, the total cost of the project is **20495,25 euros** including personnel, software and hardware costs. We have to add an overhead cost of 25 % to the total costs of the project, so finally the total number will be **25619,06 euros.**

CHAPTER 7

Installation, Configuration and Execution

This chapter explains the testbed created in this final degree project; its motivations; the installation, reset and execution process; and finally, a brief description of the software developed.

7.1 Objectives

As we have seen in the Solution Proposed (Chapter 3), this project aims to develop a network following the CROWD architecture. CROWD sets some important concepts like district and region which must be defined before to develop the architecture desired. A CROWD district is a tier with a limited but fine grain scope for short time scales composed by base stations (LTE, WiFi, APs or eNBs). A CROWD Region is another tier with a broader but coarser grain scope for long time scales.

Therefore with the goal of testing the performance of a distributed mobility managed network system, this project has designed a network environment where the DMM mobility proposals can be studied. The main idea here is to take the theoretical concepts to design a physical network and apply the solution in order to check its viability. The first requirement for the network is to be formed by at least two CROWD districts to test mobility between districts. Each district have a different gateway to access Internet and one Local Controller under the hierarchical domain of a Regional controller to manage all the districts of its region.

According to the CROWD definitions these controllers are the following:

- A CROWD Local Controller (CLC)
- A CROWD Regional Controller (CRC)

The controller logic has been developed in Python, and its functions defined and oriented to route the traffic through the network. Despite the competences of a CLC include resources management and QoS, Network Discovery, Topology detection, etc... these are out of the scope of the project. In this case, the controller implements all the control plane of the network. A device running OpenFlow controller software fills the

Flow Tables of the network's nodes in order to provide it with IPv6 Internet connection. The switches of the OpenFlow network are *Linksys WRT54GL*, a Linux-based router on which we run Pantou; and *Alix PCs*, a Linux box converted into a Linux Bridge with Access Point capabilities.

7.2 Installation

This section will explain the whole installation process of the testbed of this final degree project. The installation will cover all the different equipments of the laboratory.

7.2.1 Open vSwitch

The Open vSwitch must be installed on the *Alix* which are configured as access points nodes of the SDN. The following commands are necessary to install Open vSwitch with kernel privileges[20]:

```
1  $ sudo apt-get update && apt-get dist-upgrade
2  $ sudo apt-get install linux-headers-generic
3  $ sudo apt-get install openvswitch-switch
4  $ sudo apt-get install openvswitch-datapath-source bridge-utils
5  $ module-assistant auto-install openvswitch-datapath
```

Now we can configure Open vSwitch. First we have to create the virtual bridge. Second we add the physical interfaces and the virtual ports. Finally we have to add the IPv4 of our CROWD Local Controler (CLC) and the port:

```
1  $ ovs-vsctl add-br of-switch
2  $ ovs-vsctl add-port of-switch eth0
3  $ ovs-vsctl add-port of-switch eth1
4  $ ovs-vsctl add-port of-switch wlan0
5  $ ovs-vsctl set-controller of-switch tcp:192.168.7.100:6633
```

We can display the bridge information with to check if the bridge is configured correctly

```
1  $ ovs-dpctl show
2  $ ovs-dpctl info
```

We have to take into consideration that when we add a physical interface to the virtual bridge we will lose the access to this interface. This is important if we are connected through this interface via *ssh*.

7.2.2 Pantou

This section shows the necessary steps to load the *pantou* image into the *Linksys WRT54GL*. We have to download the latest version of *pantou* from its web site. For making the uploading process easier we have renamed the downloaded image to *code.bin*.

1. Configure the computer with an ip of the private network 192.168.1.0/24

```
1  $ sudo service network-manager stop
```

```
$ sudo ifconfig eth0 down
$ sudo ifconfig eth0 up
$ sudo ip a a 192.168.1.2/24 dev eth0
```

2. Link the computer and the Linksys with an Ethernet cable
3. Set the Linksys in failsafe mode:
   - Plug the power on
   - Hold on the reset button
   - Connect and disconnect the power cord
   - Hold the reset button and wait until the DMZ light stops blinking
   - Release the reset button
4. Install *tftp* in the computer
5. Launch *tftp* program and write the following commands:

```
binary
rexmt 1
timeout 60
trace
put code.bin
```

6. Bring the *Linksys* to failsafe mode again when the image has been loaded

7.2.3 Ryu

Every controller of the software defined network must have installed *Ryu*. For the current testbed we have to install *Ryu* on both *CLC* and in the *CRC*.

```
$ sudo apt-get install python-dev
$ sudo apt-get install libxml2-dev libxslt-dev
$ git clone git://github.com/osrg/ryu.git
$ cd ryu/
$ sudo python ./setup.py install
```

7.2.4 DMM Gateway

First we need to install the Open vSwitch application with kernel privileges.

```
$ sudo apt-get update && apt-get dist-upgrade
$ sudo apt-get install linux-headers-generic
$ sudo apt-get install openvswitch-switch
$ sudo apt-get install openvswitch-datapath-source bridge-utils
$ module-assistant auto-install openvswitch-datapath
```

Then we have to configure the *Alix* into a *Gateway*. To achieve this, we have to edit some configuration variables of the Linux Kernel. We have to uncomment the line related to IPv6 forwarding in the following file:

```
1    $ vim /etc/sysctl.conf
```

In addition, we have to set the variable accept ra to 2. This mean that the *Alix* will accept Router Advertisements, and will auto configure the selected interface with received data.

```
1    $ sysctl -w net.ipv6.conf.eth1.accept_ra=2
```

Finally we can check the new configuration with the following commands:

```
1    $ cat /proc/sys/net/ipv6/conf/all/forwarding
2    $ cat /proc/sys/net/ipv6/conf/eth1/forwarding
3    $ cat /proc/sys/net/ipv6/conf/eth1/accept_ra
```

Besides previous steps, we have to change the mac address of the interface connected to our SDN in each Gateway:

```
1    $ ip link set eth0 address aa:bb:cc:dd:ee:ff
```

7.2.5 Access Point

To convert an *Alix* into an Access Point we need the *Hostapd* script. First, the required libraries must be installed to compile *Hostapd*. Then download *Hostapd* script from the official web page[21]. Finally compile the *Hostapd* program with make.

```
1    $ sudo apt-get install libnl-dev libssl-dev
2    $ wget http://w1.fi/releases/hostapd-2.1.tar.gz
3    $ tar xzvf hostapd-2.1.tar.gz
4    $ cd hostapd-2.1/hostapd
5    $ cp defconfig .config
6    $ make
```

Besides the previous steps, we have to create a configuration file like the following:

```
1    interface=wlan0
2    ssid=ALIXAP
3    channel=6
4    hw_mode=g
5    macaddr_acl=1
6    accept_mac_file=/root/hostapd-2.1/hostapd/mac_filter
```

Accept mac file contains the absolute path to a file with the allowed macs of the experiment nodes.

7.2.6 Additional Configurations

To allow *ssh* access without writing the key in the *Alix* nodes must write the following commands on the computer in charge of the management of the SDN: (X.Y.Z.W is the IPv4 of an *Alix* node of the SDN)

```
1    $ ssh-keygen -b 4096 -t rsa
2    $ ssh-copy-id root@X.Y.Z.W
```

This is the same operation but in the *Linksys* nodes. After creating the key, access via *telnet* and configure the *ssh* service creating a password with the *passwd* command. Then log out, send the generated key with the *scp* command and connect via *ssh* to write the following commands:

```
1    $ cd /etc/dropbear
2    $ cat /tmp/id_*.pub >> authorized_keys
3    $ chmod 0600 authorized_keys
4    $ rm /etc/init.d/telnet
5    $ rm /tmp/id_rsa.pub
```

7.3 Execution

To execute the testbed we have to launch the *Ryu* controllers of the SDN. In the CROWD Regional Controller we invoke *Ryu* with the following instruction:

```
1    $ ryu-manager --verbose CRC.py
```

In our two CROWD Local Controllers we have to start *Ryu* too:

```
1    $ ryu-manager --verbose CLC.py
```

7.4 Reset

Before start doing handovers we have to reset our SDN. To simplify the process we have created a reset script which can be launched from any computer linked with the control network. Basically, the script launch these commands on each element of the SDN.

On the *Linksys* nodes:

```
1    $ /etc/init.d/openflow restart
```

On the *Alix APs* we have to restart the Open vSwitch and add two drop rules to avoid managing multicast DNS messages within the controllers of the SDN.

```
1    $ ovs-vsctl emer-reset
2    $ ovs-vsctl set-controller of-switch tcp:CLC_D1_IP:CLC_PORT
3    $ ovs-ofctl add-flow of-switch "table=0,
         dl_dst=33:33:00:00:00:16, actions=drop"
```

```
4       $ ovs-ofctl add-flow of-switch "table=0,
            dl_dst=33:33:00:00:00:fb, actions=drop"
```

Finally, we have to reset the Gateway of each district. We have to delete each tunnel created and we have to erase every forwarding rule:

```
1       $ ip -6 tu del Tun$NUMBER
2       $ ip -6 ro del table myorg
3       $ ip -6 ro flush table myorg
4       $ ip -6 ru del table myorg
```

To simplify the execution process it is possible to manage all the important nodes of the SDN with the help of the virtual terminal emulator *terminator*[22] As we can see in the image below, we can manage CRC, CLC District A and CLC district B in the same window. We can also manage every Access Point of the SDN and administer default gateways of each district.

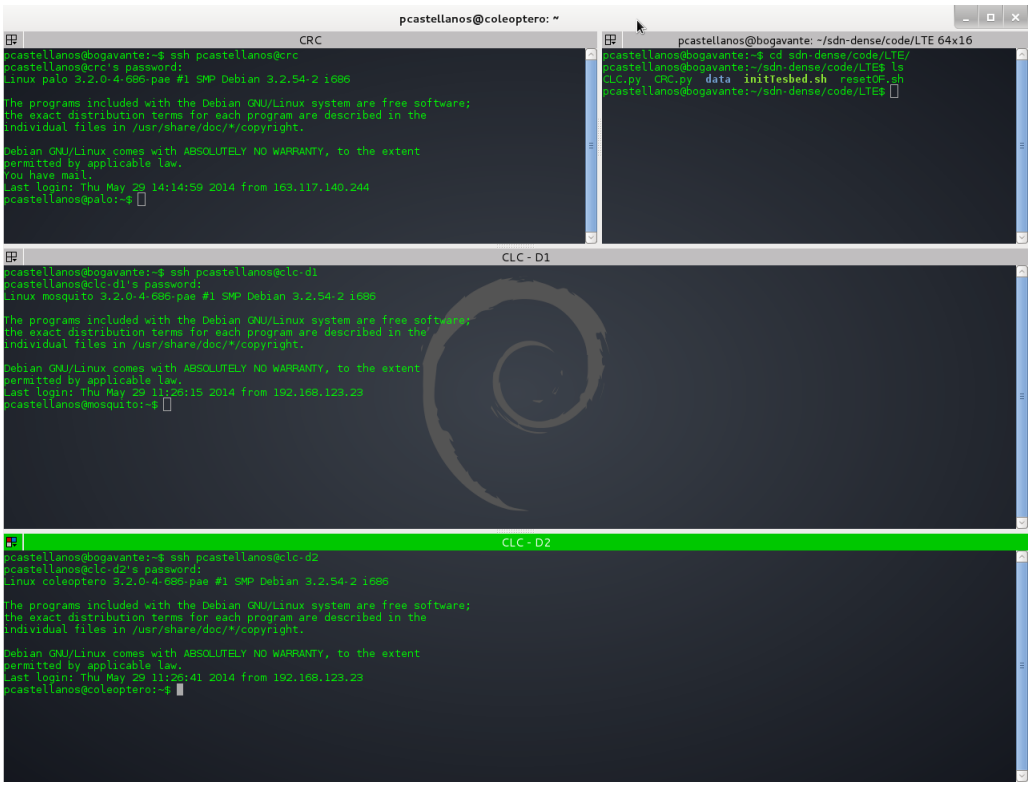


Figure 7.1: Terminator

7.5 Mobile Node Application

For the Host-Based Functionality of the Project we have developed a script with a graphic user interface that makes easier the interaction between the mobile node and the open flow network. The application has been developed with *Python* and *Pygtk* [23]. With this script we allow the user to scan the wireless access points of the SDN. The user can connect with any of these access points and with the IPv4 network. The script also allows the user to capture the traffic generated through the wireless interface. The main window displays all possible usage of the application to the user. As we can see in Figure 7.2, the user can change the wireless interface name, scan for available WiFi networks and start or stop sniffing wireless interface.

When the user clicks on the scan button, the script uses the wireless interface to display all the access points of the SDN and allows the user to connect with all the found networks as we can see in Figure 7.3.

Finally, the last window of the program shows the credits of the software application developed.

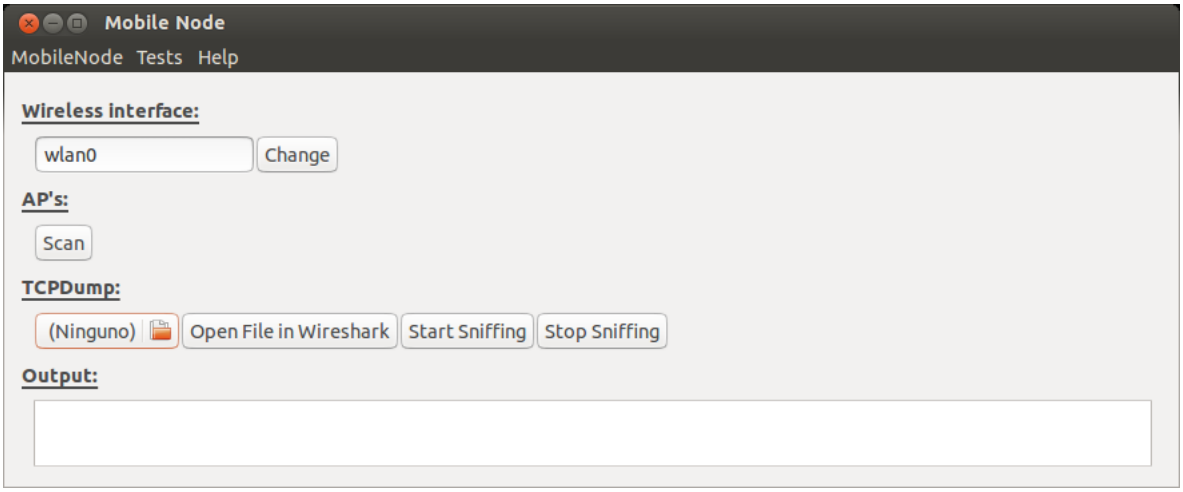


Figure 7.2: Mobile Node Main Window

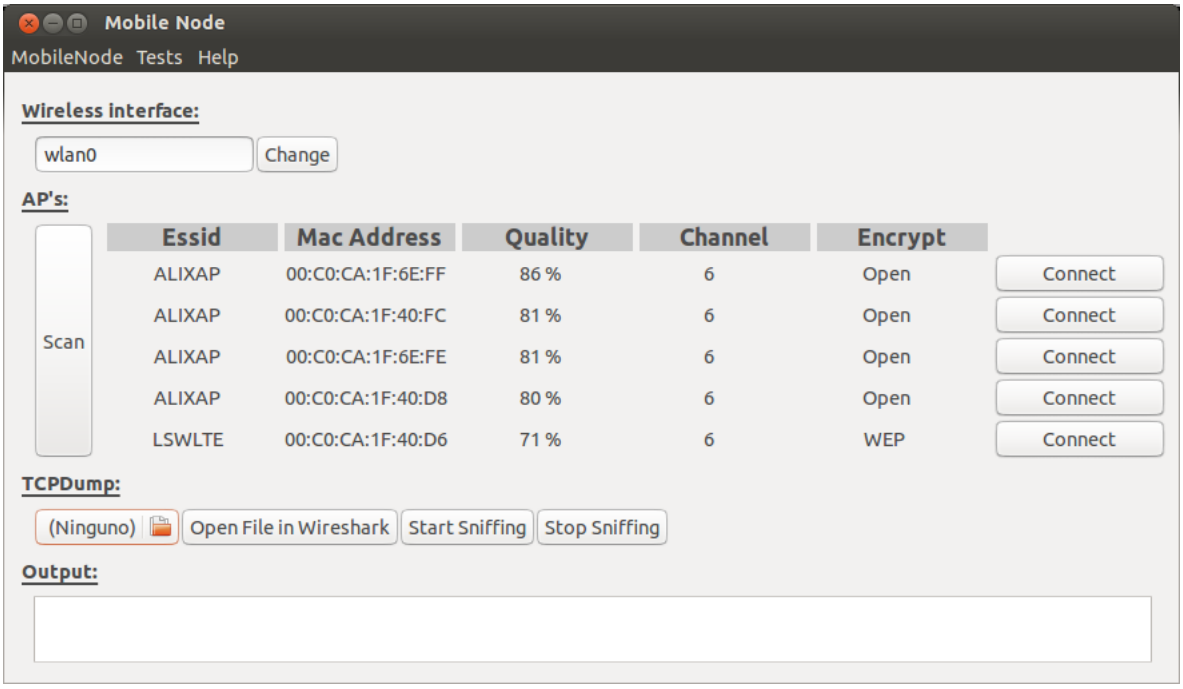


Figure 7.3: Mobile Node Scanning Wireless Networks

Figure 7.4: Mobile Node About Window

CHAPTER 8

Conclusions and future work

During the realization of this project I have studied the mobility problem of future high networks. I have also taken into consideration all solutions proposed by the DMM work group in the framework of the CROWD European Project. Once the problem and all possible solutions have been reviewed, I have developed our own way to solve the issue based on the implementation of a Software Defined Network using Open Flow as network protocol.

As is shown in the Experiments (Chapter 5) the results are not as high as we can expect but this is related with the limitations of the Open Flow controller (Ryu). This software is very new and has a long road ahead to improve. Moreover, the performance of the SDN seems to be very hardware-dependent so it is fair to think that with better and faster network nodes we should have an improvement in the performance. Also, the OpenFlow version used was the 1.0 and the 1.4 has been released so there is still much room for improvement and a wide space to optimize the software developed.

With this development I have had the opportunity to strengthen a lot of network knowledges that will improve the skills I have learnt during the realization of my course of studies in Computer Science. I have always enjoyed using open source approaches to solve engineering problems and with this final degree project, I had the opportunity to collaborate with a high skilled team contributing with a different point of view, as a computer engineer as I am.

References

[1] SDN, "Sdn wikipedia web page," 27 May 2014. Available at `http://en.wikipedia.org/wiki/Software-defined_networking`.

[2] J. Zuniga, C. Bernardos, A. de la Oliva, T. Melia, R. Costa, and A. Reznik, "Distributed mobility management: A standards landscape," *Communications Magazine, IEEE*, vol. 51, pp. 80–87, March 2013.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[4] S. Deering, Cisco, R. Hinden, and Nokia, *"Internet Protocol, Version 6 (IPv6) Specification"*. Internet Engineering Task Force (IETF). RFC 2460.

[5] I. S. Institute and U. of Southern California, *"Internet Protocol DARPA Internet Program Protocol Specification"*. Internet Engineering Task Force (IETF). RFC 791.

[6] C. Perkins, D. Johnson, and J. Arkko, *"Mobility Support in IPv6"*. Internet Engineering Task Force (IETF). RFC 6275.

[7] S. Gundavelli, K. Leung, V. Devarapalli, K. Chowdhury, and B. Patil, *"Proxy Mobile IPv6"*. Internet Engineering Task Force (IETF). RFC 5213.

[8] H. Soliman, *"Mobile IPv6 Support for Dual Stack Hosts and Routers"*. Internet Engineering Task Force (IETF). RFC 5555.

[9] D. Johnson, C. Perkins, and J. Arkko, *Mobility Support in IPv6*. Internet Engineering Task Force (IETF). RFC 3775.

[10] O. N. Foundation, "Official web page," 27 May 2014. Available at `https://www.opennetworking.org/`.

[11] T. Narten, IBM, E. Nordmark, S. Microsystems, W. Simpson, Daydreamer, H. Soliman, and E. Technologies, *"Neighbor Discovery for IP version 6 (IPv6)"*. Internet Engineering Task Force (IETF). (RFC 4861).

[12] A. Conta, Transwitch, S. Deering, C. Systems, M. Gupta, and T. Networks, *"Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification"*. Internet Engineering Task Force (IETF). RFC 4443.

[13] Alix, "Alix official web page," 23 May 2014. Available at `http://www.pcengines.ch/alix2d3.htm`.

[14] L. WRT54GL, "Linksys official wiki page," 24 May 2014. Available at http://wiki.openwrt.org/toh/linksys/wrt54g.

[15] Scappy, "Scappy official web page," 27 May 2014. Available at http://www.secdev.org/projects/scapy/.

[16] Ryu, "Ryu official web page," 27 May 2014. Available at http://osrg.github.io/ryu/.

[17] O. vSwitch, "Open vswitch official web page," 27 May 2014. Available at http://openvswitch.org/.

[18] Pantou, "Pantou official web page," 26 May 2014. Available at http://archive.openflow.org/wk/index.php/Pantou_:_OpenFlow_1.0_for_OpenWRT.

[19] CDF, "Cdf wikipedia page," 23 May 2014. Available at http://en.wikipedia.org/wiki/Cumulative_distribution_function.

[20] O. vSwitch Installation, "Official wiki," 27 May 2014. Available at http://git.openvswitch.org/cgi-bin/gitweb.cgi?p=openvswitch;a=blob_plain;f=FAQ;hb=HEAD.

[21] Hostapd, "Official hostapd web page," June 2, 2014. Available at http://wireless.kernel.org/en/users/Documentation/hostapd.

[22] Terminator, "Official terminator web page," 2 Jun 2014. Available at http://gnometerminator.blogspot.com.es/p/introduction.html.

[23] PyGTK, "Official pygtk web page," 2 Jun 2014. Available at http://www.pygtk.org/.

APPENDIX A

CRC

This section contains the documentation of the CROWD Regional Controller developed during the realization of this final degree project.

CRC

1.0

Generated by Doxygen 1.8.1.2

Wed Jun 4 2014 13:24:11

Contents

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Class Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 CRC Namespace Reference

Classes

- class MN_entry

  This class represents a mobile node element stored in the mobile node cache.
- class Tunnel_entry

  This class represents a tunnel element stored in the tunnel cache.

Variables

- tuple s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
- list MN_cache = []

  Cache with all the mobile nodes connected to the SDN.
- list Tunnel_cache = []

  Cache with all the tunnels created in the SDN.
- tuple data = conn.recv(BUFFER_SIZE)
- tuple adv_type = str(adv_type)
- tuple mn_mac = str(mn_mac)
- tuple prefix = str(prefix)
- tuple prefixlen = str(prefixlen)
- tuple default_gw = str(default_gw)
- tuple controller = str(controller)
- tuple respuesta = struct.pack('!1s39si39s4s', "N", "00", 0, "00", "AAAA")
- int in_tunnel = 0
- string old_def_gw = ""
- string CLC_IP = ""
- found = False
- origin = IP4_GW_D1
- list prefix_used = MN_cache[i]
- ipv4_MN = prefix
- string name = "Lte0"
- destiny = ipv4_MN
- string direccion = "T"
- tuple s2 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
- tuple lte_message = struct.pack('!1s4s39s39s39si17s1s','L', name, destiny, origin, prefix_used, int(prefixlen), mn_mac, direccion)

5.1.1 Variable Documentation

5.1.1.1 tuple CRC.adv_type = str(adv_type)

Definition at line 172 of file CRC.py.

5.1.1.2 string CRC.CLC_IP = ""

Definition at line 264 of file CRC.py.

5.1.1.3 tuple CRC.controller = str(controller)

Definition at line 177 of file CRC.py.

5.1.1.4 tuple CRC.data = conn.recv(BUFFER_SIZE)

Definition at line 168 of file CRC.py.

5.1.1.5 tuple CRC.default_gw = str(default_gw)

Definition at line 176 of file CRC.py.

5.1.1.6 CRC.destiny = ipv4_MN

Definition at line 290 of file CRC.py.

5.1.1.7 string CRC.direccion = "T"

Definition at line 291 of file CRC.py.

5.1.1.8 CRC.found = False

Definition at line 266 of file CRC.py.

5.1.1.9 int CRC.in_tunnel = 0

Definition at line 213 of file CRC.py.

5.1.1.10 CRC.ipv4_MN = prefix

Definition at line 286 of file CRC.py.

5.1.1.11 tuple CRC.lte_message = struct.pack('!1s4s39s39s39si17s1s','L', name, destiny, origin, prefix_used, int(prefixlen), mn_mac, direccion)

Definition at line 295 of file CRC.py.

5.1.1.12 list CRC.MN_cache = [ ]

Cache with all the mobile nodes connected to the SDN.

Definition at line 158 of file CRC.py.

5.1.1.13 tuple CRC.mn_mac = str(mn_mac)

Definition at line 173 of file CRC.py.

5.1.1.14 string CRC.name = "Lte0"

Definition at line 289 of file CRC.py.

5.1.1.15 list CRC.old_def_gw = ""

Definition at line 214 of file CRC.py.

5.1.1.16 CRC.origin = IP4_GW_D1

Definition at line 273 of file CRC.py.

5.1.1.17 tuple CRC.prefix = str(prefix)

Definition at line 174 of file CRC.py.

5.1.1.18 list CRC.prefix_used = MN_cache[i]

Definition at line 275 of file CRC.py.

5.1.1.19 list CRC.prefixlen = str(prefixlen)

Definition at line 175 of file CRC.py.

5.1.1.20 tuple CRC.respuesta = struct.pack('!1s39si39s4s', "N", "00", 0, "00", "AAAA")

Definition at line 198 of file CRC.py.

5.1.1.21 tuple CRC.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

Definition at line 7 of file CRC.py.

5.1.1.22 tuple CRC.s2 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

Definition at line 294 of file CRC.py.

5.1.1.23 list CRC.Tunnel_cache = [ ]

Cache with all the tunnels created in the SDN.

Definition at line 161 of file CRC.py.

5.2 Dijkstra Namespace Reference

Functions

- def Dijkstra

  Find shortest paths from the start vertex to all vertices nearer than or equal to the end.
- def shortestPath

  Find a single shortest path from the given start vertex to the given end vertex.

5.2.1 Function Documentation

5.2.1.1 def Dijkstra.Dijkstra (*G, start, end* = None)

Find shortest paths from the start vertex to all vertices nearer than or equal to the end.

The input graph G is assumed to have the following representation: A vertex can be any object that can be used as an index into a dictionary. G is a dictionary, indexed by vertices. For any vertex v, G[v] is itself a dictionary, indexed by the neighbors of v. For any edge v->w, G[v][w] is the length of the edge. This is related to the representation in http://www.python.org/doc/essays/graphs.html where Guido van Rossum suggests representing graphs as dictionaries mapping vertices to lists of neighbors, however dictionaries of edges have many advantages over lists: they can store extra information (here, the lengths), they support fast existence tests, and they allow easy modification of the graph by edge insertion and removal. Such modifications are not needed here but are important in other graph algorithms. Since dictionaries obey iterator protocol, a graph represented as described here could be handed without modification to an algorithm using Guido's representation.

Of course, G and G[v] need not be Python dict objects; they can be any other object that obeys dict protocol, for instance a wrapper in which vertices are URLs and a call to G[v] loads the web page and finds its links.

The output is a pair (D,P) where D[v] is the distance from start to v and P[v] is the predecessor of v along the shortest path from s to v.

Dijkstra's algorithm is only guaranteed to work correctly when all edge lengths are positive. This code does not verify this property for all edges (only the edges seen before the end vertex is reached), but will correctly compute shortest paths even for some graphs with negative edges, and will raise an exception if it discovers that a negative edge has caused it to make a mistake.

Definition at line 49 of file Dijkstra.py.

5.2.1.2 def Dijkstra.shortestPath (*G, start, end*)

Find a single shortest path from the given start vertex to the given end vertex.

The input has the same conventions as Dijkstra(). The output is a list of the vertices in order along the shortest path.

Definition at line 79 of file Dijkstra.py.

5.3 priodict Namespace Reference

Classes

- class priorityDictionary

5.4 testbedData Namespace Reference

Variables

- string CLC_A_IP = "192.168.123.1"

  IPv4 Address of the CROWD Local Controller of the District A.
- string CLC_B_IP = "192.168.123.5"

  IPv4 Address of the CROWD Local Controller of the District B.
- int CLC_PORT = 9991

  Listening Port of the CROWD Local Controller.
- int BUFFER_SIZE = 1024

  Size of the messages received from the CROWD Local Controller.
- string CRC_IP = "192.168.123.2"

  IPv4 of the CROWD Regional Controller.
- int CRC_PORT = 6633

  Listening Port of the CROWD Regional Controller.
- int TunnID = 0

  Initial Tunnel Identification.
- string GW_D1 = "2001:720:410:1001:20d:b9ff:fe2e:9855"

  IPv6 Address of the Default Gateway from the District A.
- string GW_D2 = "2001:720:410:1001:20d:b9ff:fe2e:9805"

  IPv6 Address of the Default Gateway from the District B.
- string IP4_GW_D1 = "163.117.140.71"

  IPv4 Address of the Default Gateway from the District A.
- string IP4_GW_D2 = "163.117.140.75"

  IPv4 Address of the Default Gateway from the District B.

5.4.1 Variable Documentation

5.4.1.1 int testbedData.BUFFER_SIZE = 1024

Size of the messages received from the CROWD Local Controller.

Definition at line 12 of file testbedData.py.

5.4.1.2 string testbedData.CLC_A_IP = "192.168.123.1"

IPv4 Address of the CROWD Local Controller of the District A.

Definition at line 6 of file testbedData.py.

5.4.1.3 string testbedData.CLC_B_IP = "192.168.123.5"

IPv4 Address of the CROWD Local Controller of the District B.

Definition at line 8 of file testbedData.py.

5.4.1.4 int testbedData.CLC_PORT = 9991

Listening Port of the CROWD Local Controller.

Definition at line 10 of file testbedData.py.

5.4.1.5 string testbedData.CRC_IP = "192.168.123.2"

IPv4 of the CROWD Regional Controller.

Definition at line 14 of file testbedData.py.

5.4.1.6 int testbedData.CRC_PORT = 6633

Listening Port of the CROWD Regional Controller.

Definition at line 16 of file testbedData.py.

5.4.1.7 string testbedData.GW_D1 = "2001:720:410:1001:20d:b9ff:fe2e:9855"

IPv6 Address of the Default Gateway from the District A.

Definition at line 20 of file testbedData.py.

5.4.1.8 string testbedData.GW_D2 = "2001:720:410:1001:20d:b9ff:fe2e:9805"

IPv6 Address of the Default Gateway from the District B.

Definition at line 22 of file testbedData.py.

5.4.1.9 string testbedData.IP4_GW_D1 = "163.117.140.71"

IPv4 Address of the Default Gateway from the District A.

Definition at line 24 of file testbedData.py.

5.4.1.10 string testbedData.IP4_GW_D2 = "163.117.140.75"

IPv4 Address of the Default Gateway from the District B.

Definition at line 26 of file testbedData.py.

5.4.1.11 int testbedData.TunnID = 0

Initial Tunnel Identification.

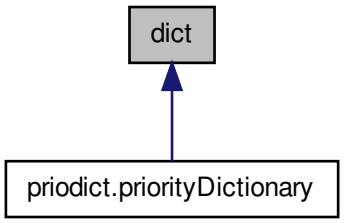
Definition at line 18 of file testbedData.py.

Chapter 6

Class Documentation

6.1 dict Class Reference

Inheritance diagram for dict:



The documentation for this class was generated from the following file:

- priodict.py

6.2 CRC.MN_entry Class Reference

This class represents a mobile node element stored in the mobile node cache.

Public Member Functions

- def __init__

  Each Mobile Node Entry has the following elements:

Public Attributes

- mac
- prefix
- mask

- def_gw

6.2.1 Detailed Description

This class represents a mobile node element stored in the mobile node cache.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 17 of file CRC.py.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 def CRC.MN_entry.__init__ (*self*, *mac* = 0, *prefix* = 0, *mask* = 0, *def_gw* = 0, *controller* = 0)

Each Mobile Node Entry has the following elements:

Parameters

mac	address Hardware Address of the wireless interface
prefix	String Prefix generated by the CLC associated with the mobile node mac address
mask	String IPv6 network mask
def_gw	String IPv6 of the default gateway of the district in which the mobile node is connected
controller	String IPv4 of the CLC associated with the Mobile Node

Definition at line 27 of file CRC.py.

6.2.3 Member Data Documentation

6.2.3.1 CRC.MN_entry.def_gw

Definition at line 31 of file CRC.py.

6.2.3.2 CRC.MN_entry.mac

Definition at line 28 of file CRC.py.

6.2.3.3 CRC.MN_entry.mask

Definition at line 30 of file CRC.py.

6.2.3.4 CRC.MN_entry.prefix

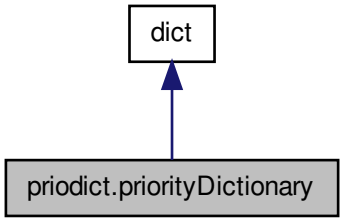
Definition at line 29 of file CRC.py.

The documentation for this class was generated from the following file:

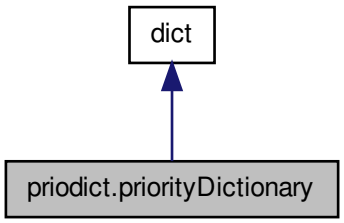
- CRC.py

6.3 priodict.priorityDictionary Class Reference

Inheritance diagram for priodict.priorityDictionary:



Collaboration diagram for priodict.priorityDictionary:



Public Member Functions

- def __init__

  Initialize priorityDictionary by creating binary heap of pairs (value,key).
- def smallest

  Find smallest item after removing deleted items from front of heap.
- def __iter__

  Create destructive sorted iterator of priorityDictionary.
- def __setitem__

  Change value stored in dictionary and add corresponding pair to heap.
- def setdefault

  *Reimplement setdefault to pass through our customized **setitem**.*

6.3.1 Detailed Description

Definition at line 14 of file priodict.py.

6.3.2 Constructor & Destructor Documentation

6.3.2.1 def priodict.priorityDictionary.__init__ (*self*)

Initialize priorityDictionary by creating binary heap of pairs (value,key).

Note that changing or removing a dict entry will not remove the old pair from the heap until it is found by smallest() or until the heap is rebuilt.

Definition at line 19 of file priodict.py.

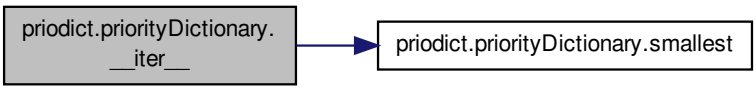
6.3.3 Member Function Documentation

6.3.3.1 def priodict.priorityDictionary.__iter__ (*self*)

Create destructive sorted iterator of priorityDictionary.

Definition at line 45 of file priodict.py.

Here is the call graph for this function:



6.3.3.2 def priodict.priorityDictionary.__setitem__ (*self, key, val*)

Change value stored in dictionary and add corresponding pair to heap.

Rebuilds the heap if the number of deleted items gets large, to avoid memory leakage.

Definition at line 56 of file priodict.py.

6.3.3.3 def priodict.priorityDictionary.setdefault (*self, key, val*)

Reimplement setdefault to pass through our customized **setitem**.

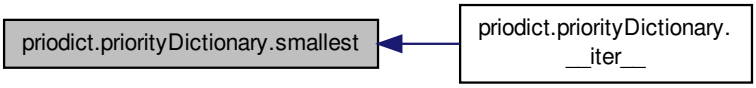
Definition at line 73 of file priodict.py.

6.3.3.4 def priodict.priorityDictionary.smallest (*self*)

Find smallest item after removing deleted items from front of heap.

Definition at line 25 of file priodict.py.

Here is the caller graph for this function:



The documentation for this class was generated from the following file:

- priodict.py

6.4 CRC.Tunnel_entry Class Reference

This class represents a tunnel element stored in the tunnel cache.

Public Member Functions

- def __init__

  Each Tunnel Entry has the following elements:

Public Attributes

- name
- origin
- destiny
- prefix
- prefixlen
- mn

6.4.1 Detailed Description

This class represents a tunnel element stored in the tunnel cache.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 39 of file CRC.py.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 def CRC.Tunnel_entry.__init__ (*self*, *name* = 0, *origin* = 0, *destiny* = 0, *prefix* = 0, *prefixlen* = 0, *mn* = 0)

Each Tunnel Entry has the following elements:

Parameters

name	String Name of the tunnel interface
origin	String IPv6 of the Gateway which is the origin of the tunnel created
destiny	String IPv6 of the Gateway which is the destiny of the tunnel created
prefix	String Prefix generated by the CLC associated with the mobile node mac address
prefixlen	String IPv6 network mask
mn	String Mac hardware address of the Mobile Node

Definition at line 50 of file CRC.py.

6.4.3 Member Data Documentation

6.4.3.1 CRC.Tunnel_entry.destiny

Definition at line 53 of file CRC.py.

6.4.3.2 CRC.Tunnel_entry.mn

Definition at line 56 of file CRC.py.

6.4.3.3 CRC.Tunnel_entry.name

Definition at line 51 of file CRC.py.

6.4.3.4 CRC.Tunnel_entry.origin

Definition at line 52 of file CRC.py.

6.4.3.5 CRC.Tunnel_entry.prefix

Definition at line 54 of file CRC.py.

6.4.3.6 CRC.Tunnel_entry.prefixlen

Definition at line 55 of file CRC.py.

The documentation for this class was generated from the following file:

- CRC.py

Chapter 7

File Documentation

7.1 CRC.py File Reference

Classes

- class CRC.MN_entry

  This class represents a mobile node element stored in the mobile node cache.
- class CRC.Tunnel_entry

  This class represents a tunnel element stored in the tunnel cache.

Namespaces

- namespace CRC

Variables

- tuple CRC.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
- list CRC.MN_cache = []

  Cache with all the mobile nodes connected to the SDN.
- list CRC.Tunnel_cache = []

  Cache with all the tunnels created in the SDN.
- tuple CRC.data = conn.recv(BUFFER_SIZE)
- tuple CRC.adv_type = str(adv_type)
- tuple CRC.mn_mac = str(mn_mac)
- tuple CRC.prefix = str(prefix)
- tuple CRC.prefixlen = str(prefixlen)
- tuple CRC.default_gw = str(default_gw)
- tuple CRC.controller = str(controller)
- tuple CRC.respuesta = struct.pack('!1s39si39s4s', "N", "00", 0, "00", "AAAA")
- int CRC.in_tunnel = 0
- string CRC.old_def_gw = ""
- string CRC.CLC_IP = ""
- CRC.found = False
- CRC.origin = IP4_GW_D1
- list CRC.prefix_used = MN_cache[i]
- CRC.ipv4_MN = prefix
- string CRC.name = "Lte0"
- CRC.destiny = ipv4_MN

- string CRC.direccion = "T"
- tuple CRC.s2 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
- tuple CRC.lte_message = struct.pack('!1s4s39s39s39si17s1s','L', name, destiny, origin, prefix_used, int(prefixlen), mn_mac, direccion)

7.2 Dijkstra.py File Reference

Namespaces

- namespace Dijkstra

Functions

- def Dijkstra.Dijkstra

  Find shortest paths from the start vertex to all vertices nearer than or equal to the end.
- def Dijkstra.shortestPath

  Find a single shortest path from the given start vertex to the given end vertex.

7.3 priodict.py File Reference

Classes

- class priodict.priorityDictionary

Namespaces

- namespace priodict

7.4 testbedData.py File Reference

Namespaces

- namespace testbedData

Variables

- string testbedData.CLC_A_IP = "192.168.123.1"

  IPv4 Address of the CROWD Local Controller of the District A.
- string testbedData.CLC_B_IP = "192.168.123.5"

  IPv4 Address of the CROWD Local Controller of the District B.
- int testbedData.CLC_PORT = 9991

  Listening Port of the CROWD Local Controller.
- int testbedData.BUFFER_SIZE = 1024

  Size of the messages received from the CROWD Local Controller.
- string testbedData.CRC_IP = "192.168.123.2"

  IPv4 of the CROWD Regional Controller.
- int testbedData.CRC_PORT = 6633

  Listening Port of the CROWD Regional Controller.

- int testbedData.TunnID = 0

  Initial Tunnel Identification.
- string testbedData.GW_D1 = "2001:720:410:1001:20d:b9ff:fe2e:9855"

  IPv6 Address of the Default Gateway from the District A.
- string testbedData.GW_D2 = "2001:720:410:1001:20d:b9ff:fe2e:9805"

  IPv6 Address of the Default Gateway from the District B.
- string testbedData.IP4_GW_D1 = "163.117.140.71"

  IPv4 Address of the Default Gateway from the District A.
- string testbedData.IP4_GW_D2 = "163.117.140.75"

  IPv4 Address of the Default Gateway from the District B.

APPENDIX B

CLC

This section contains the documentation of the CROWD Local Controller developed during the realization of this final degree project.

CLC

1.0

Generated by Doxygen 1.8.1.2

Wed Jun 4 2014 13:27:43

Contents

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Class Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 CLC Namespace Reference

Classes

- class arp_entry

  This class represents an ARP element stored in the ARP cache.
- class ipv6_entry

  This class represents an IPv6 element stored in the IPv6 cache.
- class BCE

  This class represents a tuple of elements to be stored in the cache.
- class GW

  This class represents the Default Gateway of the District.
- class network_prefix

  This class represents a network prefix.
- class SimpleSwitch

  This class represents the implementation of the CROWD Local Controller using Ryu as openflow controller.

Variables

- int ARP_REQUEST = 0x0001
- int ARP_REPLY = 0x00002
- int ETH_TYPE_8021Q = 0x8100
- int ETH_TYPE_IP = 0x0800
- TIME_LOGGING = False
- tuple district = os.getcwd()
- tuple time_file = open("debug"+district+".log", "a")

5.1.1 Variable Documentation

5.1.1.1 int CLC.ARP_REPLY = 0x00002

Definition at line 51 of file CLC.py.

5.1.1.2 int CLC.ARP_REQUEST = 0x0001

Definition at line 50 of file CLC.py.

5.1.1.3 list CLC.district = os.getcwd()

Definition at line 56 of file CLC.py.

5.1.1.4 int CLC.ETH_TYPE_8021Q = 0x8100

Definition at line 52 of file CLC.py.

5.1.1.5 int CLC.ETH_TYPE_IP = 0x0800

Definition at line 53 of file CLC.py.

5.1.1.6 tuple CLC.time_file = open("debug"+district+".log", "a")

Definition at line 60 of file CLC.py.

5.1.1.7 CLC.TIME_LOGGING = False

Definition at line 55 of file CLC.py.

5.2 Dijkstra Namespace Reference

Functions

- def Dijkstra

  Find shortest paths from the start vertex to all vertices nearer than or equal to the end.
- def shortestPath

  Find a single shortest path from the given start vertex to the given end vertex.

5.2.1 Function Documentation

5.2.1.1 def Dijkstra.Dijkstra (*G, start, end* = None)

Find shortest paths from the start vertex to all vertices nearer than or equal to the end.

The input graph G is assumed to have the following representation: A vertex can be any object that can be used as an index into a dictionary. G is a dictionary, indexed by vertices. For any vertex v, G[v] is itself a dictionary, indexed by the neighbors of v. For any edge v->w, G[v][w] is the length of the edge. This is related to the representation in http://www.python.org/doc/essays/graphs.html where Guido van Rossum suggests representing graphs as dictionaries mapping vertices to lists of neighbors, however dictionaries of edges have many advantages over lists: they can store extra information (here, the lengths), they support fast existence tests, and they allow easy modification of the graph by edge insertion and removal. Such modifications are not needed here but are important in other graph algorithms. Since dictionaries obey iterator protocol, a graph represented as described here could be handed without modification to an algorithm using Guido's representation.

Of course, G and G[v] need not be Python dict objects; they can be any other object that obeys dict protocol, for instance a wrapper in which vertices are URLs and a call to G[v] loads the web page and finds its links.

The output is a pair (D,P) where D[v] is the distance from start to v and P[v] is the predecessor of v along the shortest path from s to v.

Dijkstra's algorithm is only guaranteed to work correctly when all edge lengths are positive. This code does not verify this property for all edges (only the edges seen before the end vertex is reached), but will correctly compute

shortest paths even for some graphs with negative edges, and will raise an exception if it discovers that a negative edge has caused it to make a mistake.

Definition at line 49 of file Dijkstra.py.

5.2.1.2 def Dijkstra.shortestPath (*G, start, end*)

Find a single shortest path from the given start vertex to the given end vertex.

The input has the same conventions as Dijkstra(). The output is a list of the vertices in order along the shortest path.

Definition at line 79 of file Dijkstra.py.

5.3 priodict Namespace Reference

Classes

- class priorityDictionary

5.4 testbedData Namespace Reference

Variables

- int PREF_INIT = 2101

  Initial prefix of the District.
- string IP4_GW1 = "192.168.124.8"

  IPv4 Address of the Default Gateway from the District.
- string MAC_GW = "33:33:ff:00:00:01"

  Mac Hardware Address of the Default Gateway from the District.
- string IP6_GW1 = "fe80::20d:b9ff:fe2e:9855"

  IPv6 Local Link Address of the Default Gateway from the District.
- string FAKE_MAC_GW = "aa:bb:cc:dd:ee:ff"

  Fake Mac Address of the Default Gateway from the District.
- string DGW_IP6 = "2001:720:410:1001:20d:b9ff:fe2e:9855"

  IPv6 Address of the Default Gateway from the District.
- string DG_LL = "ffe80::20d:b9ff:fe2e:9856"

  IPv6 Local Link Address of the Default Gateway from the District.
- string DW_IP4 = "163.117.140.71"

  IPv4 Address of the Default Gateway from the District.
- string M_MCAST_RO = "33:33:ff:dd:ee:ff"

  Mac Address multicast of the Default Gateway.
- string IF_INT_GW = "eth0"

  Internal interface name of the Default Gateway to the SDN.
- string IF_EXT_GW = "eth1"

  External interface name of the Default Gateway to 163.117.140.0/24 (laboratory network)
- string CLC_IP = "192.168.123.1"

  IPv4 Address of the CROWD Local Controller of the District.
- int CLC_PORT = 9991

  Listening Port of the CROWD Local Controller.
- int BUFFER_SIZE = 1024

Size of the messages received from the CROWD Regional Controller.

- string CRC_IP = "192.168.123.2"

  IPv4 Address of the CROWD Regional Controller of the SDN.

- int CRC_PORT = 6633

  Listening Port of the CROWD Regional Controller.

- int TunnID = 0

  Initial Tunnel Identification.

- string IP_CTRL_GW = "192.168.123.77"

  IPv4 Control Address of the Default Gateway from the District.

- string UDP_IP = "192.168.123.77"

  IPv4 Control Address of the Default Gateway from the District necessary to the UDP connection.

- int UDP_PORT = 32000

  Port of the Default Gateway from the District A necessary to the UDP connection.

- string AP1_IP = "192.168.123.3"

  IPv4 Address of the Acces Point 1 of the District necessary to the paths dictionary.

- string AP2_IP = "192.168.123.4"

  IPv4 Address of the Access Point 2 of the District necessary to the paths dictionary.

- string LSW_IP = "192.168.123.7"

  IPv4 Address of the Linksys Node of the District Anecessary to the paths dictionary.

- dictionary G = {AP1_IP:{LSW_IP:1}, AP2_IP:{LSW_IP:1}, LSW_IP:{AP1_IP:1, AP2_IP:1, IP4_GW1:1}, IP4_GW1:{LSW_IP:1}}

  Paths dictionary used by the Djikstra Algorithm with cost 1.

- dictionary G_port = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP4_GW1:1}, IP4_GW1:{LSW_IP:-1}}

  Outgoing ports from current node.

- dictionary G_in_ports = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP4_GW1:1}, IP4_GW1:{LSW_IP:1}}

  Incoming ports from current node.

- dictionary G6 = {AP1_IP:{LSW_IP:1}, AP2_IP:{LSW_IP:1}, LSW_IP:{AP1_IP:1, AP2_IP:1, IP6_GW1:1}, IP6_GW1:{LSW_IP:1}}

  IPv6 dictionary paths.

- dictionary G6_port = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP6_GW1:1}, IP6_GW1:{LSW_IP:-1}}

  Outgoing ports from current node.

- dictionary G6_in_ports = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP6_GW1:1}, IP6_GW1:{LSW_IP:1}}

  Incoming ports from current node.

5.4.1 Variable Documentation

5.4.1.1 string testbedData.AP1_IP = "192.168.123.3"

IPv4 Address of the Acces Point 1 of the District necessary to the paths dictionary.

Definition at line 66 of file testbedData.py.

5.4.1.2 string testbedData.AP2_IP = "192.168.123.4"

IPv4 Address of the Access Point 2 of the District necessary to the paths dictionary.

Definition at line 69 of file testbedData.py.

5.4.1.3 int testbedData.BUFFER_SIZE = 1024

Size of the messages received from the CROWD Regional Controller.

Definition at line 45 of file testbedData.py.

5.4.1.4 string testbedData.CLC_IP = "192.168.123.1"

IPv4 Address of the CROWD Local Controller of the District.

Definition at line 39 of file testbedData.py.

5.4.1.5 int testbedData.CLC_PORT = 9991

Listening Port of the CROWD Local Controller.

Definition at line 42 of file testbedData.py.

5.4.1.6 string testbedData.CRC_IP = "192.168.123.2"

IPv4 Address of the CROWD Regional Controller of the SDN.

Definition at line 48 of file testbedData.py.

5.4.1.7 int testbedData.CRC_PORT = 6633

Listening Port of the CROWD Regional Controller.

Definition at line 51 of file testbedData.py.

5.4.1.8 string testbedData.DG_LL = "ffe80::20d:b9ff:fe2e:9856"

IPv6 Local Link Address of the Default Gateway from the District.

Definition at line 24 of file testbedData.py.

5.4.1.9 string testbedData.DGW_IP6 = "2001:720:410:1001:20d:b9ff:fe2e:9855"

IPv6 Address of the Default Gateway from the District.

Definition at line 21 of file testbedData.py.

5.4.1.10 string testbedData.DW_IP4 = "163.117.140.71"

IPv4 Address of the Default Gateway from the District.

Definition at line 27 of file testbedData.py.

5.4.1.11 string testbedData.FAKE_MAC_GW = "aa:bb:cc:dd:ee:ff"

Fake Mac Address of the Default Gateway from the District.

Definition at line 18 of file testbedData.py.

5.4.1.12 dictionary testbedData.G = {AP1_IP:{LSW_IP:1}, AP2_IP:{LSW_IP:1}, LSW_IP:{AP1_IP:1, AP2_IP:1, IP4_GW1:1}, IP4_GW1:{LSW_IP:1}}

Paths dictionary used by the Djikstra Algorithm with cost 1.

Definition at line 75 of file testbedData.py.

5.4.1.13 dictionary testbedData.G6 = {AP1_IP:{LSW_IP:1}, AP2_IP:{LSW_IP:1}, LSW_IP:{AP1_IP:1, AP2_IP:1, IP6_GW1:1}, IP6_GW1:{LSW_IP:1}}

IPv6 dictionary paths.

Definition at line 85 of file testbedData.py.

5.4.1.14 dictionary testbedData.G6_in_ports = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP6_GW1:1}, IP6_GW1:{LSW_IP:1}}

Incoming ports from current node.

Definition at line 91 of file testbedData.py.

5.4.1.15 dictionary testbedData.G6_port = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP6_GW1:1}, IP6_GW1:{LSW_IP:-1}}

Outgoing ports from current node.

Definition at line 88 of file testbedData.py.

5.4.1.16 dictionary testbedData.G_in_ports = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP4_GW1:1}, IP4_GW1:{LSW_IP:1}}

Incoming ports from current node.

Definition at line 82 of file testbedData.py.

5.4.1.17 dictionary testbedData.G_port = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP4_GW1:1}, IP4_GW1:{LSW_IP:-1}}

Outgoing ports from current node.

Definition at line 79 of file testbedData.py.

5.4.1.18 string testbedData.IF_EXT_GW = "eth1"

External interface name of the Default Gateway to 163.117.140.0/24 (laboratory network)

Definition at line 36 of file testbedData.py.

5.4.1.19 string testbedData.IF_INT_GW = "eth0"

Internal interface name of the Default Gateway to the SDN.

Definition at line 33 of file testbedData.py.

5.4.1.20 string testbedData.IP4_GW1 = ”192.168.124.8”

IPv4 Address of the Default Gateway from the District.

Definition at line 9 of file testbedData.py.

5.4.1.21 string testbedData.IP6_GW1 = ”fe80::20d:b9ff:fe2e:9855”

IPv6 Local Link Address of the Default Gateway from the District.

Definition at line 15 of file testbedData.py.

5.4.1.22 string testbedData.IP_CTRL_GW = ”192.168.123.77”

IPv4 Control Address of the Default Gateway from the District.

Definition at line 57 of file testbedData.py.

5.4.1.23 string testbedData.LSW_IP = ”192.168.123.7”

IPv4 Address of the Linksys Node of the District Anecessary to the paths dictionary.

Definition at line 72 of file testbedData.py.

5.4.1.24 string testbedData.M_MCAST_RO = ”33:33:ff:dd:ee:ff”

Mac Address multicast of the Default Gateway.

Definition at line 30 of file testbedData.py.

5.4.1.25 string testbedData.MAC_GW = ”33:33:ff:00:00:01”

Mac Hardware Address of the Default Gateway from the District.

Definition at line 12 of file testbedData.py.

5.4.1.26 int testbedData.PREF_INIT = 2101

Initial prefix of the District.

Definition at line 6 of file testbedData.py.

5.4.1.27 int testbedData.TunnID = 0

Initial Tunnel Identification.

Definition at line 54 of file testbedData.py.

5.4.1.28 string testbedData.UDP_IP = ”192.168.123.77”

IPv4 Control Address of the Default Gateway from the District necessary to the UDP connection.

Definition at line 60 of file testbedData.py.

5.4.1.29 int testbedData.UDP_PORT = 32000

Port of the Default Gateway from the District A necessary to the UDP connection.

Definition at line 63 of file testbedData.py.

Chapter 6

Class Documentation

6.1 CLC.arp_entry Class Reference

This class represents an ARP element stored in the ARP cache.

Public Member Functions

- def __init__

  Each ARP Entry has the following elements:

Public Attributes

- ip
- mac

6.1.1 Detailed Description

This class represents an ARP element stored in the ARP cache.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 68 of file CLC.py.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 def CLC.arp_entry.__init__ (*self*, *mac* = 0, *ip* = 0)

Each ARP Entry has the following elements:

Parameters

mac	int MAC address
ip	int IPv4

Definition at line 75 of file CLC.py.

6.1.3 Member Data Documentation

6.1.3.1 CLC.arp_entry.ip

Definition at line 76 of file CLC.py.

6.1.3.2 CLC.arp_entry.mac

Definition at line 77 of file CLC.py.

The documentation for this class was generated from the following file:

- CLC.py

6.2 CLC.BCE Class Reference

This class represents a tuple of elements to be stored in the cache.

Public Member Functions

- def __init__

  Each BCE object has the following elements:

Public Attributes

- MN_mac
- AP_IP
- Mcast_mac
- MN_IPv6

6.2.1 Detailed Description

This class represents a tuple of elements to be stored in the cache.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 104 of file CLC.py.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 def CLC.BCE.__init__ (*self*, *MN_mac* = 0, *AP_IP* = 0, *Mcast_mac* = 0, *MN_IPv6* = 0)

Each BCE object has the following elements:

Parameters

MN_mac	int MAC address of the mobile node
AP_IP	int IPv4 of the Access point
Mcast_mac	int Multicast MAC address of the mobile node
MN_IPv6	int IPv6 of the mobile node

Definition at line 113 of file CLC.py.

6.2.3 Member Data Documentation

6.2.3.1 CLC.BCE.AP_IP

Definition at line 115 of file CLC.py.

6.2.3.2 CLC.BCE.Mcast_mac

Definition at line 116 of file CLC.py.

6.2.3.3 CLC.BCE.MN_IPv6

Definition at line 117 of file CLC.py.

6.2.3.4 CLC.BCE.MN_mac

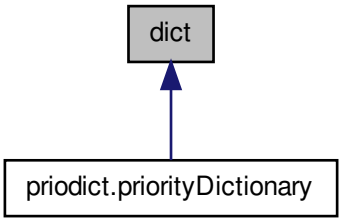
Definition at line 114 of file CLC.py.

The documentation for this class was generated from the following file:

- CLC.py

6.3 dict Class Reference

Inheritance diagram for dict:



The documentation for this class was generated from the following file:

- priodict.py

6.4 CLC.GW Class Reference

This class represents the Default Gateway of the District.

Public Member Functions

- def __init__

  Each GW object has the following elements:

Public Attributes

- GW_mac
- GW_mcast_mac
- GW_ip4
- GW_ip6
- GW_default
- GW_name

6.4.1 Detailed Description

This class represents the Default Gateway of the District.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 126 of file CLC.py.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 def CLC.GW.__init__ (*self*, *GW_mac* = 0, *GW_mcast_mac* = 0, *GW_ip4* = 0, *GW_ip6* = 0, *GW_default* = 0, *GW_name* = 0)

Each GW object has the following elements:

Parameters

self.GW_mac	String MAC address of the Gateway
self.GW_mcast_mac	String Multicast MAC address of the Gateway
self.GW_ip4	String IPv4 of the Gateway
self.GW_ip6	String IPv6 of the Gateway
self.GW_default	String With default value
self.GW_name	String Identification name of the Gateway

Definition at line 137 of file CLC.py.

6.4.3 Member Data Documentation

6.4.3.1 CLC.GW.GW_default

Definition at line 142 of file CLC.py.

6.4.3.2 CLC.GW.GW_ip4

Definition at line 140 of file CLC.py.

6.4.3.3 CLC.GW.GW_ip6

Definition at line 141 of file CLC.py.

6.4.3.4 CLC.GW.GW_mac

Definition at line 138 of file CLC.py.

6.4.3.5 CLC.GW.GW_mcast_mac

Definition at line 139 of file CLC.py.

6.4.3.6 CLC.GW.GW_name

Definition at line 143 of file CLC.py.

The documentation for this class was generated from the following file:

- CLC.py

6.5 CLC.ipv6_entry Class Reference

This class represents an IPv6 element stored in the IPv6 cache.

Public Member Functions

- def __init__

  Each IPv6 Entry has the following elements:

Public Attributes

- ipv6
- mac

6.5.1 Detailed Description

This class represents an IPv6 element stored in the IPv6 cache.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 86 of file CLC.py.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 def CLC.ipv6_entry.__init__ (*self*, *mac* = 0, *ipv6* = 0)

Each IPv6 Entry has the following elements:

Parameters

mac	int MAC address
ipv6	int IPv6

Definition at line 93 of file CLC.py.

6.5.3 Member Data Documentation

6.5.3.1 CLC.ipv6_entry.ipv6

Definition at line 94 of file CLC.py.

6.5.3.2 CLC.ipv6_entry.mac

Definition at line 95 of file CLC.py.

The documentation for this class was generated from the following file:

- CLC.py

6.6 CLC.network_prefix Class Reference

This class represents a network prefix.

Public Member Functions

- def __init__

  Each BCE object has the following elements:

Public Attributes

- prefix
- mask
- used

6.6.1 Detailed Description

This class represents a network prefix.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 152 of file CLC.py.

6.6.2 Constructor & Destructor Documentation

6.6.2.1 def CLC.network_prefix.__init__ (*self*, *prefix* = `0`, *mask* = `0`, *used* = `False`)

Each BCE object has the following elements:

Parameters

prefix	String Network prefix
mask	String Network mask
used	Boolean True if the prefix is assigned or False if not

Definition at line 160 of file CLC.py.

6.6.3 Member Data Documentation

6.6.3.1 CLC.network_prefix.mask

Definition at line 162 of file CLC.py.

6.6.3.2 CLC.network_prefix.prefix

Definition at line 161 of file CLC.py.

6.6.3.3 CLC.network_prefix.used

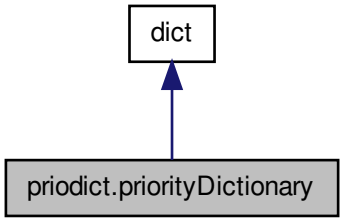
Definition at line 163 of file CLC.py.

The documentation for this class was generated from the following file:

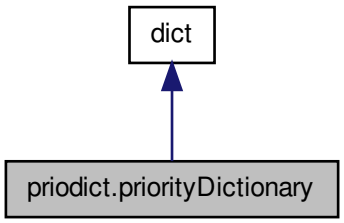
- CLC.py

6.7 priodict.priorityDictionary Class Reference

Inheritance diagram for priodict.priorityDictionary:



Collaboration diagram for priodict.priorityDictionary:



Public Member Functions

- def __init__

  Initialize priorityDictionary by creating binary heap of pairs (value,key).
- def smallest

  Find smallest item after removing deleted items from front of heap.
- def __iter__

  Create destructive sorted iterator of priorityDictionary.
- def __setitem__

  Change value stored in dictionary and add corresponding pair to heap.
- def setdefault

  *Reimplement setdefault to pass through our customized **setitem**.*

6.7.1 Detailed Description

Definition at line 14 of file priodict.py.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 def priodict.priorityDictionary.__init__ (*self*)

Initialize priorityDictionary by creating binary heap of pairs (value,key).

Note that changing or removing a dict entry will not remove the old pair from the heap until it is found by smallest() or until the heap is rebuilt.

Definition at line 19 of file priodict.py.

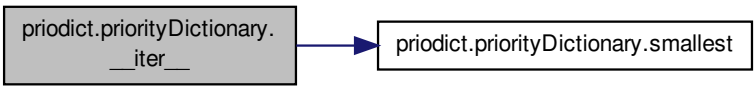
6.7.3 Member Function Documentation

6.7.3.1 def priodict.priorityDictionary.__iter__ (*self*)

Create destructive sorted iterator of priorityDictionary.

Definition at line 45 of file priodict.py.

Here is the call graph for this function:



6.7.3.2 def priodict.priorityDictionary.__setitem__ (*self, key, val*)

Change value stored in dictionary and add corresponding pair to heap.

Rebuilds the heap if the number of deleted items gets large, to avoid memory leakage.

Definition at line 56 of file priodict.py.

6.7.3.3 def priodict.priorityDictionary.setdefault (*self, key, val*)

Reimplement setdefault to pass through our customized **setitem**.

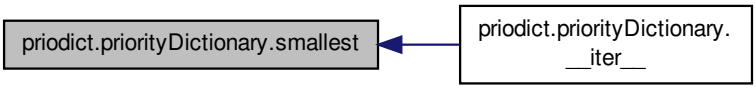
Definition at line 73 of file priodict.py.

6.7.3.4 def priodict.priorityDictionary.smallest (*self*)

Find smallest item after removing deleted items from front of heap.

Definition at line 25 of file priodict.py.

Here is the caller graph for this function:

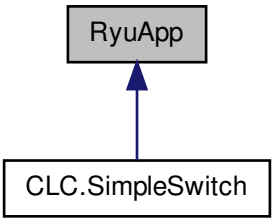


The documentation for this class was generated from the following file:

- priodict.py

6.8 RyuApp Class Reference

Inheritance diagram for RyuApp:



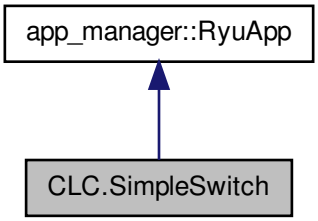
The documentation for this class was generated from the following file:

- CLC.py

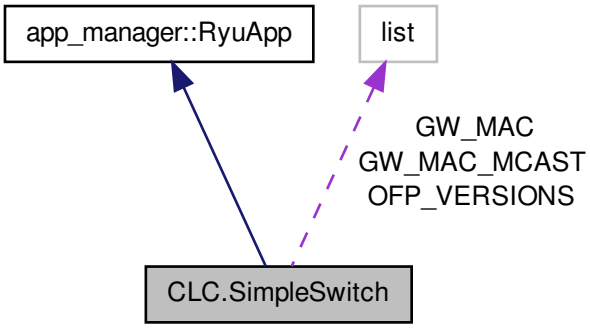
6.9 CLC.SimpleSwitch Class Reference

This class represents the implementation of the CROWD Local Controller using Ryu as openflow controller.

Inheritance diagram for CLC.SimpleSwitch:



Collaboration diagram for CLC.SimpleSwitch:



Public Member Functions

- def __init__

  This function is launched in a thread to manage all the messages received from the CROWD Regional Controller.
- def remove_flow_entries_from_ip

  This function deletes all openflow rules of a SDN node identified by its IP.
- def find_APs_to_remove

  This function deletes all openflow rules of a SDN node identified by its IP.
- def remove_old_entries

  This function deletes all old openflow rules from a datapath identified by its mac address.
- def remove_flow_entries_from_ip2

  This function deletes all openflow rules of a SDN node identified by its IP.
- def remove_old_entries2

  This function deletes all old openflow rules from a datapath identified by its mac address.
- def push_ACTION

  This function sets a new open flow rule entry in the openvswitch database.
- def echo_request_handler

This function sens a request message to the openflow SDN.

- def ipv4_to_int

  This function converts a IPv4 to an integer.

- def forge_arp

  This function creates an ARP message.

- def answer_arp

  This function answers an ARP message.

Public Attributes

- arp

  ARP object for the Default Gateway of the district.

- arp_cache

  ARP cache table.

- ipv6

  IPv6 object representing the Default Gateway.

- ipv6_cache

  IPv6 cache table.

- bce_table

  BCE cache table.

- dpset
- GW_default

  Default Gateway Object created with its MAC, MCAST MAC, IPv4 and IPv6 addresses.

- GW_table

  Gateway cache table.

- lock
- Pool

  Pool of prefixes used in this district.

- G

  Dictionary of SDN nodes (IPv4)

- G_port
- G_in_ports
- G6

  Dictionary of SDN nodes (IPv6)

- G6_port
- G6_in_ports

Static Public Attributes

- list OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]

  Open flow protocol version managed by ryu to execute this code.

- list GW_MAC = [FAKE_MAC_GW]

  This fake MAC must be the same in each district of the SDN.

- list GW_MAC_MCAST = [M_MCAST_RO]

  This fake Mcast MAC must be the same in each district of the SDN.

6.9.1 Detailed Description

This class represents the implementation of the CROWD Local Controller using Ryu as openflow controller.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 172 of file CLC.py.

6.9.2 Constructor & Destructor Documentation

6.9.2.1 def CLC.SimpleSwitch.__init__ (*self, args, kwargs*)

This function is launched in a thread to manage all the messages received from the CROWD Regional Controller.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 188 of file CLC.py.

6.9.3 Member Function Documentation

6.9.3.1 def CLC.SimpleSwitch.answer_arp (*self, data, gw_ip_addr, gw_mac_addr*)

This function answers an ARP message.

Parameters

data	String Message
gw_ip_addr	String IP address of the GW
gw_mac_addr	String Mac address of the GW

Author

Pablo Castellanos

Returns

ARP message

Date

03/06/2014

Definition at line 1243 of file CLC.py.

Here is the call graph for this function:

6.9.3.2 def CLC.SimpleSwitch.echo_request_handler (*self, ev*)

This function sens a request message to the openflow SDN.

Parameters

ev	String Open Flow Message

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 1162 of file CLC.py.

6.9.3.3 def CLC.SimpleSwitch.find_APs_to_remove (*self, path_new, path_old*)

This function deletes all openflow rules of a SDN node identified by its IP.

Parameters

path_new	List with the new path
path_old	List with the old path

Returns

path_to_remove with the old Access Points

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 859 of file CLC.py.

6.9.3.4 def CLC.SimpleSwitch.forge_arp (*self, gw_mac_addr, dst_mac_addr, gw_ip_addr, dst_ip_addr*)

This function creates an ARP message.

Parameters

gw_mac_addr	String Mac address of the GW
dst_mac_addr	String Mac address of the MN
gw_ip_addr	String IP address of the GW
dst_ip_addr	String IP address of the MN

Author

Pablo Castellanos

Returns

ARP message

Date

03/06/2014

Definition at line 1219 of file CLC.py.

Here is the caller graph for this function:

6.9.3.5 def CLC.SimpleSwitch.ipv4_to_int (*self, string*)

This function converts a IPv4 to an integer.

Parameters

string	String IPv4

Returns

int IPv4 without dots

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 1180 of file CLC.py.

6.9.3.6 def CLC.SimpleSwitch.push_ACTION (*self, dp, in_port, out_port, src_mac_addr, dst_mac_addr*)

This function sets a new open flow rule entry in the openvswitch database.

Parameters

dp	List with the new path
in_port	int Incoming Open Flow port
out_port	int Outgoing Open Flow port
src_mac_addr	String with source mac address
dst_mac_addr	String with destination mac address

Author

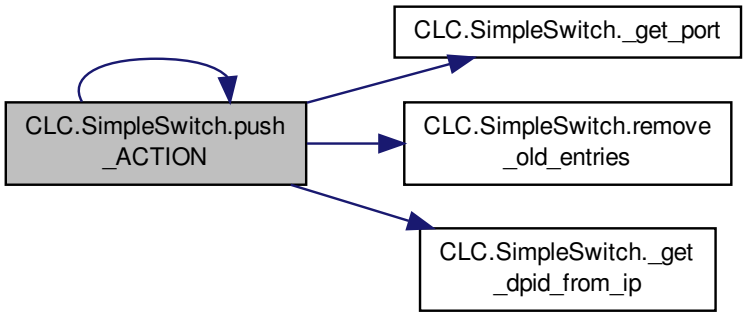
Pablo Castellanos

Date

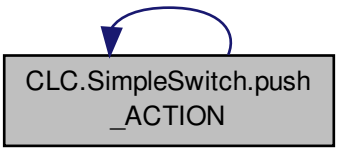
03/06/2014

Definition at line 981 of file CLC.py.

Here is the call graph for this function:



Here is the caller graph for this function:



6.9.3.7 def CLC.SimpleSwitch.remove_flow_entries_from_ip (*self, path_to_remove, src_mac_addr, dst_mac_addr*)

This function deletes all openflow rules of a SDN node identified by its IP.

Author

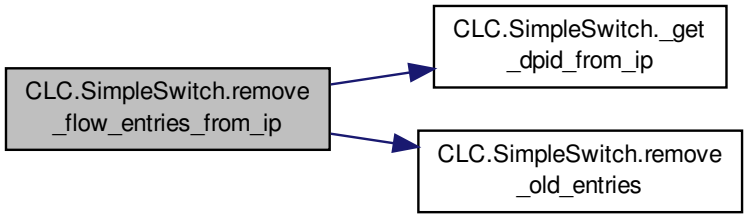
Pablo Castellanos

Date

03/06/2014

Definition at line 841 of file CLC.py.

Here is the call graph for this function:



6.9.3.8 def CLC.SimpleSwitch.remove_flow_entries_from_ip2 (*self, path_to_remove, src_mac_addr*)

This function deletes all openflow rules of a SDN node identified by its IP.

Author

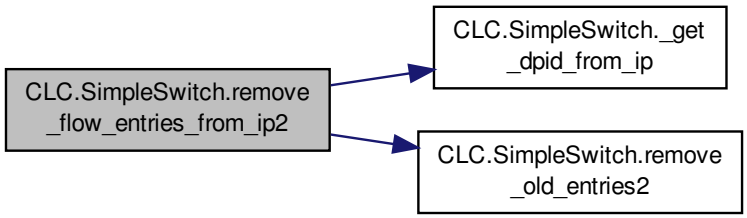
Pablo Castellanos

Date

03/06/2014

Definition at line 925 of file CLC.py.

Here is the call graph for this function:



6.9.3.9 def CLC.SimpleSwitch.remove_old_entries (*self, dp, src_mac_addr, dst_mac_addr*)

This function deletes all old openflow rules from a datapath identified by its mac address.

Parameters

dp	List with the new path
src_mac_addr	String with source mac address
dst_mac_addr	String with destination mac address

Author

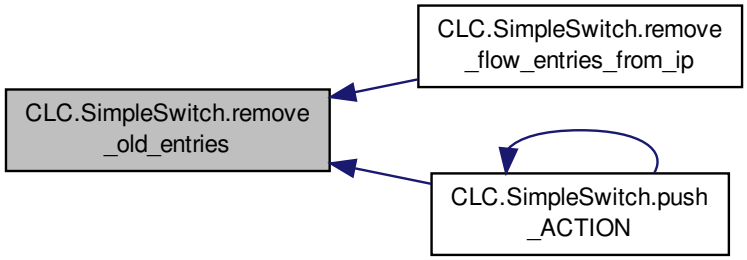
Pablo Castellanos

Date

03/06/2014

Definition at line 884 of file CLC.py.

Here is the caller graph for this function:



6.9.3.10 def CLC.SimpleSwitch.remove_old_entries2 (*self, dp, src_mac_addr*)

This function deletes all old openflow rules from a datapath identified by its mac address.

Parameters

dp	List with the new path
src_mac_addr	String with source mac address
dst_mac_addr	String with destination mac address

Author

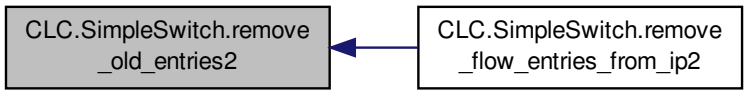
Pablo Castellanos

Date

03/06/2014

Definition at line 943 of file CLC.py.

Here is the caller graph for this function:



6.9.4 Member Data Documentation

6.9.4.1 CLC.SimpleSwitch.arp

ARP object for the Default Gateway of the district.

Formed with its MAC and its IPv4 address

Definition at line 191 of file CLC.py.

6.9.4.2 CLC.SimpleSwitch.arp_cache

ARP cache table.

Definition at line 193 of file CLC.py.

6.9.4.3 CLC.SimpleSwitch.bce_table

BCE cache table.

Definition at line 199 of file CLC.py.

6.9.4.4 CLC.SimpleSwitch.dpset

Definition at line 200 of file CLC.py.

6.9.4.5 CLC.SimpleSwitch.G

Dictionary of SDN nodes (IPv4)

Definition at line 212 of file CLC.py.

6.9.4.6 CLC.SimpleSwitch.G6

Dictionary of SDN nodes (IPv6)

Definition at line 217 of file CLC.py.

6.9.4.7 CLC.SimpleSwitch.G6_in_ports

Definition at line 219 of file CLC.py.

6.9.4.8 CLC.SimpleSwitch.G6_port

Definition at line 218 of file CLC.py.

6.9.4.9 CLC.SimpleSwitch.G_in_ports

Definition at line 214 of file CLC.py.

6.9.4.10 CLC.SimpleSwitch.G_port

Definition at line 213 of file CLC.py.

6.9.4.11 CLC.SimpleSwitch.GW_default

Default Gateway Object created with its MAC, MCAST MAC, IPv4 and IPv6 addresses.

Definition at line 202 of file CLC.py.

6.9.4.12 list CLC.SimpleSwitch.GW_MAC = [FAKE_MAC_GW] [static]

This fake MAC must be the same in each district of the SDN.

Definition at line 182 of file CLC.py.

6.9.4.13 list CLC.SimpleSwitch.GW_MAC_MCAST = [M_MCAST_RO] [static]

This fake Mcast MAC must be the same in each district of the SDN.

Definition at line 185 of file CLC.py.

6.9.4.14 CLC.SimpleSwitch.GW_table

Gateway cache table.

Definition at line 204 of file CLC.py.

6.9.4.15 CLC.SimpleSwitch.ipv6

IPv6 object representing the Default Gateway.

Formed with its MAC and its IPv6 address

Definition at line 195 of file CLC.py.

6.9.4.16 CLC.SimpleSwitch.ipv6_cache

IPv6 cache table.

Definition at line 197 of file CLC.py.

6.9.4.17 CLC.SimpleSwitch.lock

Definition at line 205 of file CLC.py.

6.9.4.18 list CLC.SimpleSwitch.OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION] [static]

Open flow protocol version managed by ryu to execute this code.

Definition at line 178 of file CLC.py.

6.9.4.19 CLC.SimpleSwitch.Pool

Pool of prefixes used in this district.

Definition at line 207 of file CLC.py.

The documentation for this class was generated from the following file:

- CLC.py

Chapter 7

File Documentation

7.1 CLC.py File Reference

Classes

- class CLC.arp_entry

  This class represents an ARP element stored in the ARP cache.
- class CLC.ipv6_entry

  This class represents an IPv6 element stored in the IPv6 cache.
- class CLC.BCE

  This class represents a tuple of elements to be stored in the cache.
- class CLC.GW

  This class represents the Default Gateway of the District.
- class CLC.network_prefix

  This class represents a network prefix.
- class CLC.SimpleSwitch

  This class represents the implementation of the CROWD Local Controller using Ryu as openflow controller.

Namespaces

- namespace CLC

Variables

- int CLC.ARP_REQUEST = 0x0001
- int CLC.ARP_REPLY = 0x00002
- int CLC.ETH_TYPE_8021Q = 0x8100
- int CLC.ETH_TYPE_IP = 0x0800
- CLC.TIME_LOGGING = False
- tuple CLC.district = os.getcwd()
- tuple CLC.time_file = open("debug"+district+".log", "a")

7.2 Dijkstra.py File Reference

Namespaces

- namespace Dijkstra

Functions

- def Dijkstra.Dijkstra

  Find shortest paths from the start vertex to all vertices nearer than or equal to the end.
- def Dijkstra.shortestPath

  Find a single shortest path from the given start vertex to the given end vertex.

7.3 priodict.py File Reference

Classes

- class priodict.priorityDictionary

Namespaces

- namespace priodict

7.4 testbedData.py File Reference

Namespaces

- namespace testbedData

Variables

- int testbedData.PREF_INIT = 2101

  Initial prefix of the District.
- string testbedData.IP4_GW1 = "192.168.124.8"

  IPv4 Address of the Default Gateway from the District.
- string testbedData.MAC_GW = "33:33:ff:00:00:01"

  Mac Hardware Address of the Default Gateway from the District.
- string testbedData.IP6_GW1 = "fe80::20d:b9ff:fe2e:9855"

  IPv6 Local Link Address of the Default Gateway from the District.
- string testbedData.FAKE_MAC_GW = "aa:bb:cc:dd:ee:ff"

  Fake Mac Address of the Default Gateway from the District.
- string testbedData.DGW_IP6 = "2001:720:410:1001:20d:b9ff:fe2e:9855"

  IPv6 Address of the Default Gateway from the District.
- string testbedData.DG_LL = "ffe80::20d:b9ff:fe2e:9856"

  IPv6 Local Link Address of the Default Gateway from the District.
- string testbedData.DW_IP4 = "163.117.140.71"

  IPv4 Address of the Default Gateway from the District.
- string testbedData.M_MCAST_RO = "33:33:ff:dd:ee:ff"

  Mac Address multicast of the Default Gateway.
- string testbedData.IF_INT_GW = "eth0"

  Internal interface name of the Default Gateway to the SDN.
- string testbedData.IF_EXT_GW = "eth1"

  External interface name of the Default Gateway to 163.117.140.0/24 (laboratory network)
- string testbedData.CLC_IP = "192.168.123.1"

  IPv4 Address of the CROWD Local Controller of the District.

- int testbedData.CLC_PORT = 9991

  Listening Port of the CROWD Local Controller.
- int testbedData.BUFFER_SIZE = 1024

  Size of the messages received from the CROWD Regional Controller.
- string testbedData.CRC_IP = "192.168.123.2"

  IPv4 Address of the CROWD Regional Controller of the SDN.
- int testbedData.CRC_PORT = 6633

  Listening Port of the CROWD Regional Controller.
- int testbedData.TunnID = 0

  Initial Tunnel Identification.
- string testbedData.IP_CTRL_GW = "192.168.123.77"

  IPv4 Control Address of the Default Gateway from the District.
- string testbedData.UDP_IP = "192.168.123.77"

  IPv4 Control Address of the Default Gateway from the District necessary to the UDP connection.
- int testbedData.UDP_PORT = 32000

  Port of the Default Gateway from the District A necessary to the UDP connection.
- string testbedData.AP1_IP = "192.168.123.3"

  IPv4 Address of the Acces Point 1 of the District necessary to the paths dictionary.
- string testbedData.AP2_IP = "192.168.123.4"

  IPv4 Address of the Access Point 2 of the District necessary to the paths dictionary.
- string testbedData.LSW_IP = "192.168.123.7"

  IPv4 Address of the Linksys Node of the District Anecessary to the paths dictionary.
- dictionary testbedData.G = {AP1_IP:{LSW_IP:1}, AP2_IP:{LSW_IP:1}, LSW_IP:{AP1_IP:1, AP2_IP:1, IP4_GW1:1}, IP4_GW1:{LSW_IP:1}}

  Paths dictionary used by the Djikstra Algorithm with cost 1.
- dictionary testbedData.G_port = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP4_GW1:1}, IP4_GW1:{LSW_IP:-1}}

  Outgoing ports from current node.
- dictionary testbedData.G_in_ports = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP4_GW1:1}, IP4_GW1:{LSW_IP:1}}

  Incoming ports from current node.
- dictionary testbedData.G6 = {AP1_IP:{LSW_IP:1}, AP2_IP:{LSW_IP:1}, LSW_IP:{AP1_IP:1, AP2_IP:1, IP6_GW1:1}, IP6_GW1:{LSW_IP:1}}

  IPv6 dictionary paths.
- dictionary testbedData.G6_port = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP6_GW1:1}, IP6_GW1:{LSW_IP:-1}}

  Outgoing ports from current node.
- dictionary testbedData.G6_in_ports = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP6_GW1:1}, IP6_GW1:{LSW_IP:1}}

  Incoming ports from current node.

APPENDIX C

MobileNode

This section contains the documentation of the Mobile Node script developed during the realization of this final degree project.

CLC

1.0

Generated by Doxygen 1.8.1.2

Wed Jun 4 2014 13:27:43

Contents

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Class Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 CLC Namespace Reference

Classes

- class arp_entry

  This class represents an ARP element stored in the ARP cache.
- class ipv6_entry

  This class represents an IPv6 element stored in the IPv6 cache.
- class BCE

  This class represents a tuple of elements to be stored in the cache.
- class GW

  This class represents the Default Gateway of the District.
- class network_prefix

  This class represents a network prefix.
- class SimpleSwitch

  This class represents the implementation of the CROWD Local Controller using Ryu as openflow controller.

Variables

- int ARP_REQUEST = 0x0001
- int ARP_REPLY = 0x00002
- int ETH_TYPE_8021Q = 0x8100
- int ETH_TYPE_IP = 0x0800
- TIME_LOGGING = False
- tuple district = os.getcwd()
- tuple time_file = open("debug"+district+".log", "a")

5.1.1 Variable Documentation

5.1.1.1 int CLC.ARP_REPLY = 0x00002

Definition at line 51 of file CLC.py.

5.1.1.2 int CLC.ARP_REQUEST = 0x0001

Definition at line 50 of file CLC.py.

5.1.1.3 list CLC.district = os.getcwd()

Definition at line 56 of file CLC.py.

5.1.1.4 int CLC.ETH_TYPE_8021Q = 0x8100

Definition at line 52 of file CLC.py.

5.1.1.5 int CLC.ETH_TYPE_IP = 0x0800

Definition at line 53 of file CLC.py.

5.1.1.6 tuple CLC.time_file = open("debug"+district+".log", "a")

Definition at line 60 of file CLC.py.

5.1.1.7 CLC.TIME_LOGGING = False

Definition at line 55 of file CLC.py.

5.2 Dijkstra Namespace Reference

Functions

- def Dijkstra

  Find shortest paths from the start vertex to all vertices nearer than or equal to the end.
- def shortestPath

  Find a single shortest path from the given start vertex to the given end vertex.

5.2.1 Function Documentation

5.2.1.1 def Dijkstra.Dijkstra (*G, start, end* = None)

Find shortest paths from the start vertex to all vertices nearer than or equal to the end.

The input graph G is assumed to have the following representation: A vertex can be any object that can be used as an index into a dictionary. G is a dictionary, indexed by vertices. For any vertex v, G[v] is itself a dictionary, indexed by the neighbors of v. For any edge v->w, G[v][w] is the length of the edge. This is related to the representation in http://www.python.org/doc/essays/graphs.html where Guido van Rossum suggests representing graphs as dictionaries mapping vertices to lists of neighbors, however dictionaries of edges have many advantages over lists: they can store extra information (here, the lengths), they support fast existence tests, and they allow easy modification of the graph by edge insertion and removal. Such modifications are not needed here but are important in other graph algorithms. Since dictionaries obey iterator protocol, a graph represented as described here could be handed without modification to an algorithm using Guido's representation.

Of course, G and G[v] need not be Python dict objects; they can be any other object that obeys dict protocol, for instance a wrapper in which vertices are URLs and a call to G[v] loads the web page and finds its links.

The output is a pair (D,P) where D[v] is the distance from start to v and P[v] is the predecessor of v along the shortest path from s to v.

Dijkstra's algorithm is only guaranteed to work correctly when all edge lengths are positive. This code does not verify this property for all edges (only the edges seen before the end vertex is reached), but will correctly compute

shortest paths even for some graphs with negative edges, and will raise an exception if it discovers that a negative edge has caused it to make a mistake.

Definition at line 49 of file Dijkstra.py.

5.2.1.2 def Dijkstra.shortestPath (*G, start, end*)

Find a single shortest path from the given start vertex to the given end vertex.

The input has the same conventions as Dijkstra(). The output is a list of the vertices in order along the shortest path.

Definition at line 79 of file Dijkstra.py.

5.3 priodict Namespace Reference

Classes

- class priorityDictionary

5.4 testbedData Namespace Reference

Variables

- int PREF_INIT = 2101

  Initial prefix of the District.
- string IP4_GW1 = "192.168.124.8"

  IPv4 Address of the Default Gateway from the District.
- string MAC_GW = "33:33:ff:00:00:01"

  Mac Hardware Address of the Default Gateway from the District.
- string IP6_GW1 = "fe80::20d:b9ff:fe2e:9855"

  IPv6 Local Link Address of the Default Gateway from the District.
- string FAKE_MAC_GW = "aa:bb:cc:dd:ee:ff"

  Fake Mac Address of the Default Gateway from the District.
- string DGW_IP6 = "2001:720:410:1001:20d:b9ff:fe2e:9855"

  IPv6 Address of the Default Gateway from the District.
- string DG_LL = "ffe80::20d:b9ff:fe2e:9856"

  IPv6 Local Link Address of the Default Gateway from the District.
- string DW_IP4 = "163.117.140.71"

  IPv4 Address of the Default Gateway from the District.
- string M_MCAST_RO = "33:33:ff:dd:ee:ff"

  Mac Address multicast of the Default Gateway.
- string IF_INT_GW = "eth0"

  Internal interface name of the Default Gateway to the SDN.
- string IF_EXT_GW = "eth1"

  External interface name of the Default Gateway to 163.117.140.0/24 (laboratory network)
- string CLC_IP = "192.168.123.1"

  IPv4 Address of the CROWD Local Controller of the District.
- int CLC_PORT = 9991

  Listening Port of the CROWD Local Controller.
- int BUFFER_SIZE = 1024

Size of the messages received from the CROWD Regional Controller.

- string CRC_IP = "192.168.123.2"

  IPv4 Address of the CROWD Regional Controller of the SDN.

- int CRC_PORT = 6633

  Listening Port of the CROWD Regional Controller.

- int TunnID = 0

  Initial Tunnel Identification.

- string IP_CTRL_GW = "192.168.123.77"

  IPv4 Control Address of the Default Gateway from the District.

- string UDP_IP = "192.168.123.77"

  IPv4 Control Address of the Default Gateway from the District necessary to the UDP connection.

- int UDP_PORT = 32000

  Port of the Default Gateway from the District A necessary to the UDP connection.

- string AP1_IP = "192.168.123.3"

  IPv4 Address of the Acces Point 1 of the District necessary to the paths dictionary.

- string AP2_IP = "192.168.123.4"

  IPv4 Address of the Access Point 2 of the District necessary to the paths dictionary.

- string LSW_IP = "192.168.123.7"

  IPv4 Address of the Linksys Node of the District Anecessary to the paths dictionary.

- dictionary G = {AP1_IP:{LSW_IP:1}, AP2_IP:{LSW_IP:1}, LSW_IP:{AP1_IP:1, AP2_IP:1, IP4_GW1:1}, IP4_GW1:{LSW_IP:1}}

  Paths dictionary used by the Djikstra Algorithm with cost 1.

- dictionary G_port = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP4_GW1:1}, IP4_GW1:{LSW_IP:-1}}

  Outgoing ports from current node.

- dictionary G_in_ports = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP4_GW1:1}, IP4_GW1:{LSW_IP:1}}

  Incoming ports from current node.

- dictionary G6 = {AP1_IP:{LSW_IP:1}, AP2_IP:{LSW_IP:1}, LSW_IP:{AP1_IP:1, AP2_IP:1, IP6_GW1:1}, IP6_GW1:{LSW_IP:1}}

  IPv6 dictionary paths.

- dictionary G6_port = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP6_GW1:1}, IP6_GW1:{LSW_IP:-1}}

  Outgoing ports from current node.

- dictionary G6_in_ports = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP6_GW1:1}, IP6_GW1:{LSW_IP:1}}

  Incoming ports from current node.

5.4.1 Variable Documentation

5.4.1.1 string testbedData.AP1_IP = "192.168.123.3"

IPv4 Address of the Acces Point 1 of the District necessary to the paths dictionary.

Definition at line 66 of file testbedData.py.

5.4.1.2 string testbedData.AP2_IP = "192.168.123.4"

IPv4 Address of the Access Point 2 of the District necessary to the paths dictionary.

Definition at line 69 of file testbedData.py.

5.4.1.3 int testbedData.BUFFER_SIZE = 1024

Size of the messages received from the CROWD Regional Controller.

Definition at line 45 of file testbedData.py.

5.4.1.4 string testbedData.CLC_IP = "192.168.123.1"

IPv4 Address of the CROWD Local Controller of the District.

Definition at line 39 of file testbedData.py.

5.4.1.5 int testbedData.CLC_PORT = 9991

Listening Port of the CROWD Local Controller.

Definition at line 42 of file testbedData.py.

5.4.1.6 string testbedData.CRC_IP = "192.168.123.2"

IPv4 Address of the CROWD Regional Controller of the SDN.

Definition at line 48 of file testbedData.py.

5.4.1.7 int testbedData.CRC_PORT = 6633

Listening Port of the CROWD Regional Controller.

Definition at line 51 of file testbedData.py.

5.4.1.8 string testbedData.DG_LL = "ffe80::20d:b9ff:fe2e:9856"

IPv6 Local Link Address of the Default Gateway from the District.

Definition at line 24 of file testbedData.py.

5.4.1.9 string testbedData.DGW_IP6 = "2001:720:410:1001:20d:b9ff:fe2e:9855"

IPv6 Address of the Default Gateway from the District.

Definition at line 21 of file testbedData.py.

5.4.1.10 string testbedData.DW_IP4 = "163.117.140.71"

IPv4 Address of the Default Gateway from the District.

Definition at line 27 of file testbedData.py.

5.4.1.11 string testbedData.FAKE_MAC_GW = "aa:bb:cc:dd:ee:ff"

Fake Mac Address of the Default Gateway from the District.

Definition at line 18 of file testbedData.py.

5.4.1.12 dictionary testbedData.G = {AP1_IP:{LSW_IP:1}, AP2_IP:{LSW_IP:1}, LSW_IP:{AP1_IP:1, AP2_IP:1, IP4_GW1:1}, IP4_GW1:{LSW_IP:1}}

Paths dictionary used by the Djikstra Algorithm with cost 1.

Definition at line 75 of file testbedData.py.

5.4.1.13 dictionary testbedData.G6 = {AP1_IP:{LSW_IP:1}, AP2_IP:{LSW_IP:1}, LSW_IP:{AP1_IP:1, AP2_IP:1, IP6_GW1:1}, IP6_GW1:{LSW_IP:1}}

IPv6 dictionary paths.

Definition at line 85 of file testbedData.py.

5.4.1.14 dictionary testbedData.G6_in_ports = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP6_GW1:1}, IP6_GW1:{LSW_IP:1}}

Incoming ports from current node.

Definition at line 91 of file testbedData.py.

5.4.1.15 dictionary testbedData.G6_port = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP6_GW1:1}, IP6_GW1:{LSW_IP:-1}}

Outgoing ports from current node.

Definition at line 88 of file testbedData.py.

5.4.1.16 dictionary testbedData.G_in_ports = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP4_GW1:1}, IP4_GW1:{LSW_IP:1}}

Incoming ports from current node.

Definition at line 82 of file testbedData.py.

5.4.1.17 dictionary testbedData.G_port = {AP1_IP:{LSW_IP:3}, AP2_IP:{LSW_IP:2}, LSW_IP:{AP1_IP:4, AP2_IP:2, IP4_GW1:1}, IP4_GW1:{LSW_IP:-1}}

Outgoing ports from current node.

Definition at line 79 of file testbedData.py.

5.4.1.18 string testbedData.IF_EXT_GW = "eth1"

External interface name of the Default Gateway to 163.117.140.0/24 (laboratory network)

Definition at line 36 of file testbedData.py.

5.4.1.19 string testbedData.IF_INT_GW = "eth0"

Internal interface name of the Default Gateway to the SDN.

Definition at line 33 of file testbedData.py.

5.4.1.20 string testbedData.IP4_GW1 = ”192.168.124.8”

IPv4 Address of the Default Gateway from the District.

Definition at line 9 of file testbedData.py.

5.4.1.21 string testbedData.IP6_GW1 = ”fe80::20d:b9ff:fe2e:9855”

IPv6 Local Link Address of the Default Gateway from the District.

Definition at line 15 of file testbedData.py.

5.4.1.22 string testbedData.IP_CTRL_GW = ”192.168.123.77”

IPv4 Control Address of the Default Gateway from the District.

Definition at line 57 of file testbedData.py.

5.4.1.23 string testbedData.LSW_IP = ”192.168.123.7”

IPv4 Address of the Linksys Node of the District Anecessary to the paths dictionary.

Definition at line 72 of file testbedData.py.

5.4.1.24 string testbedData.M_MCAST_RO = ”33:33:ff:dd:ee:ff”

Mac Address multicast of the Default Gateway.

Definition at line 30 of file testbedData.py.

5.4.1.25 string testbedData.MAC_GW = ”33:33:ff:00:00:01”

Mac Hardware Address of the Default Gateway from the District.

Definition at line 12 of file testbedData.py.

5.4.1.26 int testbedData.PREF_INIT = 2101

Initial prefix of the District.

Definition at line 6 of file testbedData.py.

5.4.1.27 int testbedData.TunnID = 0

Initial Tunnel Identification.

Definition at line 54 of file testbedData.py.

5.4.1.28 string testbedData.UDP_IP = ”192.168.123.77”

IPv4 Control Address of the Default Gateway from the District necessary to the UDP connection.

Definition at line 60 of file testbedData.py.

5.4.1.29 int testbedData.UDP_PORT = 32000

Port of the Default Gateway from the District A necessary to the UDP connection.

Definition at line 63 of file testbedData.py.

Chapter 6

Class Documentation

6.1 CLC.arp_entry Class Reference

This class represents an ARP element stored in the ARP cache.

Public Member Functions

- def __init__

  Each ARP Entry has the following elements:

Public Attributes

- ip
- mac

6.1.1 Detailed Description

This class represents an ARP element stored in the ARP cache.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 68 of file CLC.py.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 def CLC.arp_entry.__init__ (*self*, *mac* = 0, *ip* = 0)

Each ARP Entry has the following elements:

Parameters

mac	int MAC address
ip	int IPv4

Definition at line 75 of file CLC.py.

6.1.3 Member Data Documentation

6.1.3.1 CLC.arp_entry.ip

Definition at line 76 of file CLC.py.

6.1.3.2 CLC.arp_entry.mac

Definition at line 77 of file CLC.py.

The documentation for this class was generated from the following file:

- CLC.py

6.2 CLC.BCE Class Reference

This class represents a tuple of elements to be stored in the cache.

Public Member Functions

- def __init__

  Each BCE object has the following elements:

Public Attributes

- MN_mac
- AP_IP
- Mcast_mac
- MN_IPv6

6.2.1 Detailed Description

This class represents a tuple of elements to be stored in the cache.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 104 of file CLC.py.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 def CLC.BCE.__init__ (*self*, *MN_mac* = 0, *AP_IP* = 0, *Mcast_mac* = 0, *MN_IPv6* = 0)

Each BCE object has the following elements:

Parameters

MN_mac	int MAC address of the mobile node
AP_IP	int IPv4 of the Access point
Mcast_mac	int Multicast MAC address of the mobile node
MN_IPv6	int IPv6 of the mobile node

Definition at line 113 of file CLC.py.

6.2.3 Member Data Documentation

6.2.3.1 CLC.BCE.AP_IP

Definition at line 115 of file CLC.py.

6.2.3.2 CLC.BCE.Mcast_mac

Definition at line 116 of file CLC.py.

6.2.3.3 CLC.BCE.MN_IPv6

Definition at line 117 of file CLC.py.

6.2.3.4 CLC.BCE.MN_mac

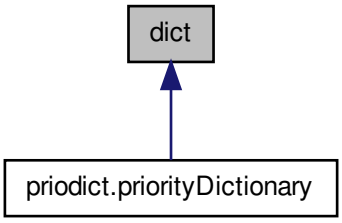
Definition at line 114 of file CLC.py.

The documentation for this class was generated from the following file:

- CLC.py

6.3 dict Class Reference

Inheritance diagram for dict:



The documentation for this class was generated from the following file:

- priodict.py

6.4 CLC.GW Class Reference

This class represents the Default Gateway of the District.

Public Member Functions

- def __init__

  Each GW object has the following elements:

Public Attributes

- GW_mac
- GW_mcast_mac
- GW_ip4
- GW_ip6
- GW_default
- GW_name

6.4.1 Detailed Description

This class represents the Default Gateway of the District.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 126 of file CLC.py.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 def CLC.GW.__init__ (*self*, *GW_mac* = 0, *GW_mcast_mac* = 0, *GW_ip4* = 0, *GW_ip6* = 0, *GW_default* = 0, *GW_name* = 0)

Each GW object has the following elements:

Parameters

self.GW_mac	String MAC address of the Gateway
self.GW_mcast_mac	String Multicast MAC address of the Gateway
self.GW_ip4	String IPv4 of the Gateway
self.GW_ip6	String IPv6 of the Gateway
self.GW_default	String With default value
self.GW_name	String Identification name of the Gateway

Definition at line 137 of file CLC.py.

6.4.3 Member Data Documentation

6.4.3.1 CLC.GW.GW_default

Definition at line 142 of file CLC.py.

6.4.3.2 CLC.GW.GW_ip4

Definition at line 140 of file CLC.py.

6.4.3.3 CLC.GW.GW_ip6

Definition at line 141 of file CLC.py.

6.4.3.4 CLC.GW.GW_mac

Definition at line 138 of file CLC.py.

6.4.3.5 CLC.GW.GW_mcast_mac

Definition at line 139 of file CLC.py.

6.4.3.6 CLC.GW.GW_name

Definition at line 143 of file CLC.py.

The documentation for this class was generated from the following file:

- CLC.py

6.5 CLC.ipv6_entry Class Reference

This class represents an IPv6 element stored in the IPv6 cache.

Public Member Functions

- def __init__

  Each IPv6 Entry has the following elements:

Public Attributes

- ipv6
- mac

6.5.1 Detailed Description

This class represents an IPv6 element stored in the IPv6 cache.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 86 of file CLC.py.

6.5.2 Constructor & Destructor Documentation

6.5.2.1 def CLC.ipv6_entry.__init__ (*self*, *mac* = 0, *ipv6* = 0)

Each IPv6 Entry has the following elements:

Parameters

mac	int MAC address
ipv6	int IPv6

Definition at line 93 of file CLC.py.

6.5.3 Member Data Documentation

6.5.3.1 CLC.ipv6_entry.ipv6

Definition at line 94 of file CLC.py.

6.5.3.2 CLC.ipv6_entry.mac

Definition at line 95 of file CLC.py.

The documentation for this class was generated from the following file:

- CLC.py

6.6 CLC.network_prefix Class Reference

This class represents a network prefix.

Public Member Functions

- def __init__

  Each BCE object has the following elements:

Public Attributes

- prefix
- mask
- used

6.6.1 Detailed Description

This class represents a network prefix.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 152 of file CLC.py.

6.6.2 Constructor & Destructor Documentation

6.6.2.1 def CLC.network_prefix.__init__ (*self*, *prefix* = `0`, *mask* = `0`, *used* = `False`)

Each BCE object has the following elements:

Parameters

prefix	String Network prefix
mask	String Network mask
used	Boolean True if the prefix is assigned or False if not

Definition at line 160 of file CLC.py.

6.6.3 Member Data Documentation

6.6.3.1 CLC.network_prefix.mask

Definition at line 162 of file CLC.py.

6.6.3.2 CLC.network_prefix.prefix

Definition at line 161 of file CLC.py.

6.6.3.3 CLC.network_prefix.used

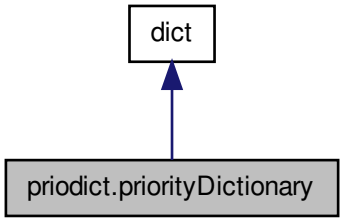
Definition at line 163 of file CLC.py.

The documentation for this class was generated from the following file:

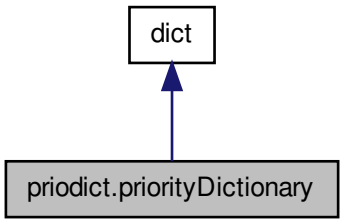
- CLC.py

6.7 priodict.priorityDictionary Class Reference

Inheritance diagram for priodict.priorityDictionary:



Collaboration diagram for priodict.priorityDictionary:



Public Member Functions

- def __init__

  Initialize priorityDictionary by creating binary heap of pairs (value,key).
- def smallest

  Find smallest item after removing deleted items from front of heap.
- def __iter__

  Create destructive sorted iterator of priorityDictionary.
- def __setitem__

  Change value stored in dictionary and add corresponding pair to heap.
- def setdefault

  *Reimplement setdefault to pass through our customized **setitem**.*

6.7.1 Detailed Description

Definition at line 14 of file priodict.py.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 def priodict.priorityDictionary.__init__ (*self*)

Initialize priorityDictionary by creating binary heap of pairs (value,key).

Note that changing or removing a dict entry will not remove the old pair from the heap until it is found by smallest() or until the heap is rebuilt.

Definition at line 19 of file priodict.py.

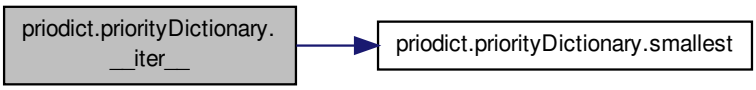
6.7.3 Member Function Documentation

6.7.3.1 def priodict.priorityDictionary.__iter__ (*self*)

Create destructive sorted iterator of priorityDictionary.

Definition at line 45 of file priodict.py.

Here is the call graph for this function:



6.7.3.2 def priodict.priorityDictionary.__setitem__ (*self, key, val*)

Change value stored in dictionary and add corresponding pair to heap.

Rebuilds the heap if the number of deleted items gets large, to avoid memory leakage.

Definition at line 56 of file priodict.py.

6.7.3.3 def priodict.priorityDictionary.setdefault (*self, key, val*)

Reimplement setdefault to pass through our customized **setitem**.

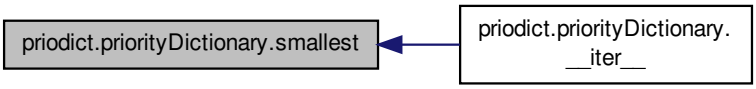
Definition at line 73 of file priodict.py.

6.7.3.4 def priodict.priorityDictionary.smallest (*self*)

Find smallest item after removing deleted items from front of heap.

Definition at line 25 of file priodict.py.

Here is the caller graph for this function:

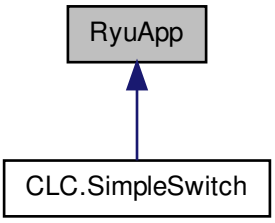


The documentation for this class was generated from the following file:

- priodict.py

6.8 RyuApp Class Reference

Inheritance diagram for RyuApp:



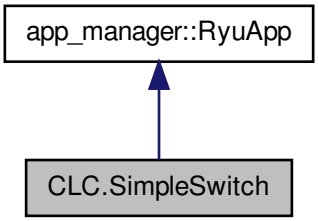
The documentation for this class was generated from the following file:

- CLC.py

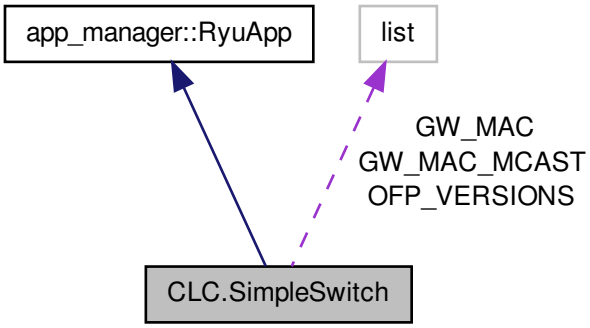
6.9 CLC.SimpleSwitch Class Reference

This class represents the implementation of the CROWD Local Controller using Ryu as openflow controller.

Inheritance diagram for CLC.SimpleSwitch:



Collaboration diagram for CLC.SimpleSwitch:



Public Member Functions

- def __init__

  This function is launched in a thread to manage all the messages received from the CROWD Regional Controller.
- def remove_flow_entries_from_ip

  This function deletes all openflow rules of a SDN node identified by its IP.
- def find_APs_to_remove

  This function deletes all openflow rules of a SDN node identified by its IP.
- def remove_old_entries

  This function deletes all old openflow rules from a datapath identified by its mac address.
- def remove_flow_entries_from_ip2

  This function deletes all openflow rules of a SDN node identified by its IP.
- def remove_old_entries2

  This function deletes all old openflow rules from a datapath identified by its mac address.
- def push_ACTION

  This function sets a new open flow rule entry in the openvswitch database.
- def echo_request_handler

This function sens a request message to the openflow SDN.

- def ipv4_to_int

  This function converts a IPv4 to an integer.

- def forge_arp

  This function creates an ARP message.

- def answer_arp

  This function answers an ARP message.

Public Attributes

- arp

  ARP object for the Default Gateway of the district.

- arp_cache

  ARP cache table.

- ipv6

  IPv6 object representing the Default Gateway.

- ipv6_cache

  IPv6 cache table.

- bce_table

  BCE cache table.

- dpset
- GW_default

  Default Gateway Object created with its MAC, MCAST MAC, IPv4 and IPv6 addresses.

- GW_table

  Gateway cache table.

- lock
- Pool

  Pool of prefixes used in this district.

- G

  Dictionary of SDN nodes (IPv4)

- G_port
- G_in_ports
- G6

  Dictionary of SDN nodes (IPv6)

- G6_port
- G6_in_ports

Static Public Attributes

- list OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]

  Open flow protocol version managed by ryu to execute this code.

- list GW_MAC = [FAKE_MAC_GW]

  This fake MAC must be the same in each district of the SDN.

- list GW_MAC_MCAST = [M_MCAST_RO]

  This fake Mcast MAC must be the same in each district of the SDN.

6.9.1 Detailed Description

This class represents the implementation of the CROWD Local Controller using Ryu as openflow controller.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 172 of file CLC.py.

6.9.2 Constructor & Destructor Documentation

6.9.2.1 def CLC.SimpleSwitch.__init__ (*self, args, kwargs*)

This function is launched in a thread to manage all the messages received from the CROWD Regional Controller.

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 188 of file CLC.py.

6.9.3 Member Function Documentation

6.9.3.1 def CLC.SimpleSwitch.answer_arp (*self, data, gw_ip_addr, gw_mac_addr*)

This function answers an ARP message.

Parameters

data	String Message
gw_ip_addr	String IP address of the GW
gw_mac_addr	String Mac address of the GW

Author

Pablo Castellanos

Returns

ARP message

Date

03/06/2014

Definition at line 1243 of file CLC.py.

Here is the call graph for this function:

6.9.3.2 def CLC.SimpleSwitch.echo_request_handler (*self, ev*)

This function sens a request message to the openflow SDN.

Parameters

ev	String Open Flow Message

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 1162 of file CLC.py.

6.9.3.3 def CLC.SimpleSwitch.find_APs_to_remove (*self, path_new, path_old*)

This function deletes all openflow rules of a SDN node identified by its IP.

Parameters

path_new	List with the new path
path_old	List with the old path

Returns

path_to_remove with the old Access Points

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 859 of file CLC.py.

6.9.3.4 def CLC.SimpleSwitch.forge_arp (*self, gw_mac_addr, dst_mac_addr, gw_ip_addr, dst_ip_addr*)

This function creates an ARP message.

Parameters

gw_mac_addr	String Mac address of the GW
dst_mac_addr	String Mac address of the MN
gw_ip_addr	String IP address of the GW
dst_ip_addr	String IP address of the MN

Author

Pablo Castellanos

Returns

ARP message

Date

03/06/2014

Definition at line 1219 of file CLC.py.

Here is the caller graph for this function:

6.9.3.5 def CLC.SimpleSwitch.ipv4_to_int (*self, string*)

This function converts a IPv4 to an integer.

Parameters

string	String IPv4

Returns

int IPv4 without dots

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 1180 of file CLC.py.

6.9.3.6 def CLC.SimpleSwitch.push_ACTION (*self, dp, in_port, out_port, src_mac_addr, dst_mac_addr*)

This function sets a new open flow rule entry in the openvswitch database.

Parameters

dp	List with the new path
in_port	int Incoming Open Flow port
out_port	int Outgoing Open Flow port
src_mac_addr	String with source mac address
dst_mac_addr	String with destination mac address

Author

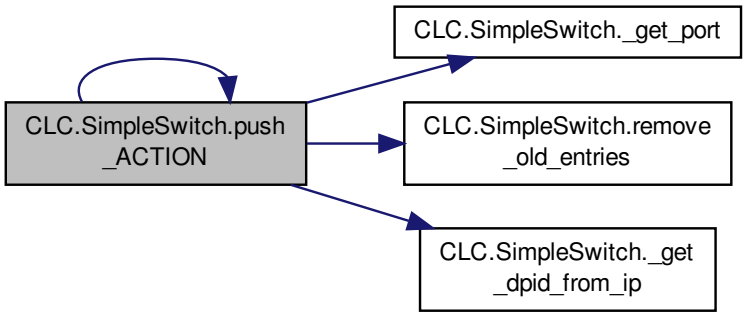
Pablo Castellanos

Date

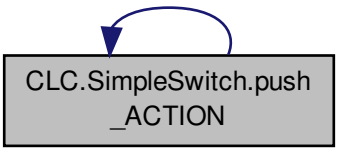
03/06/2014

Definition at line 981 of file CLC.py.

Here is the call graph for this function:



Here is the caller graph for this function:



6.9.3.7 def CLC.SimpleSwitch.remove_flow_entries_from_ip (*self, path_to_remove, src_mac_addr, dst_mac_addr*)

This function deletes all openflow rules of a SDN node identified by its IP.

Author

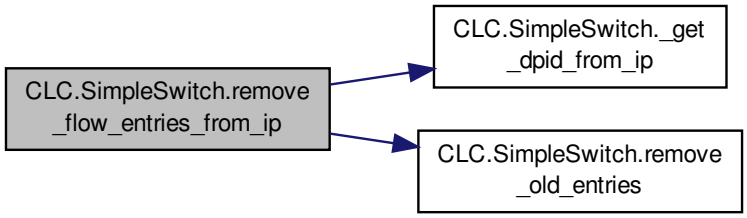
Pablo Castellanos

Date

03/06/2014

Definition at line 841 of file CLC.py.

Here is the call graph for this function:



6.9.3.8 def CLC.SimpleSwitch.remove_flow_entries_from_ip2 (*self, path_to_remove, src_mac_addr*)

This function deletes all openflow rules of a SDN node identified by its IP.

Author

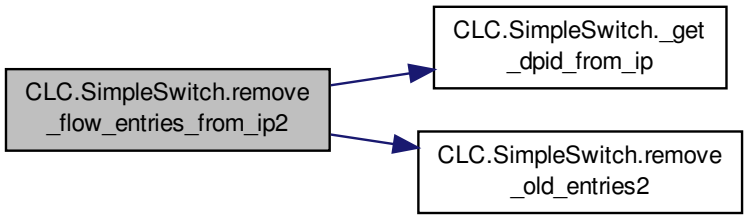
Pablo Castellanos

Date

03/06/2014

Definition at line 925 of file CLC.py.

Here is the call graph for this function:



6.9.3.9 def CLC.SimpleSwitch.remove_old_entries (*self, dp, src_mac_addr, dst_mac_addr*)

This function deletes all old openflow rules from a datapath identified by its mac address.

Parameters

dp	List with the new path
src_mac_addr	String with source mac address
dst_mac_addr	String with destination mac address

Author

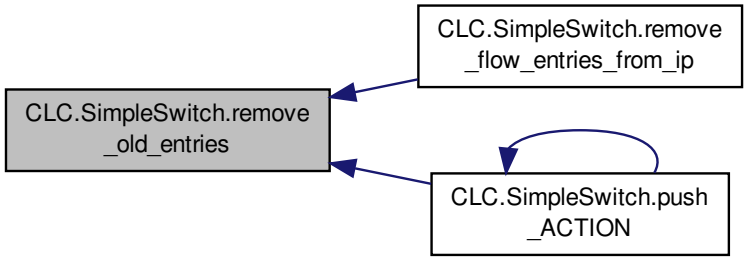
Pablo Castellanos

Date

03/06/2014

Definition at line 884 of file CLC.py.

Here is the caller graph for this function:



6.9.3.10 def CLC.SimpleSwitch.remove_old_entries2 (*self, dp, src_mac_addr*)

This function deletes all old openflow rules from a datapath identified by its mac address.

Parameters

dp	List with the new path
src_mac_addr	String with source mac address
dst_mac_addr	String with destination mac address

Author

Pablo Castellanos

Date

03/06/2014

Definition at line 943 of file CLC.py.